# VX-REXX Programmer's Guide

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Table of Contents

# Introduction

Watcom VX-REXX is an application development system for OS/2 REXX that provides visual design and programming tools to help you create OS/2 applications that have a graphical user interface. An application can be saved as an executable file so that others can use your application without requiring a copy of VX-REXX . The development system can also be used to create macro or script files to be used with other applications.

This chapter outlines how to use this manual to help you learn about VX-REXX . It also includes a list of additional sources of information that may help you in developing applications.

## What you should know before starting

This manual assumes you have installed and used OS/2. This includes :

oHow to use a mouse. You should know that OS/2 uses both mouse buttons 1 and 2 and how to use a mouse to click, double-click and drag objects.

oHow to open, close and move a window.

oHow to open a menu and choose items.

oHow to select items from a list.

oHow to use an OS/2 notebook to change settings.

oHow to use a folder.

oHow to use a template.

If you are not familiar with these topics you should browse the OS/2 Glossary or try the OS/2 Tutorial. The glossary and tutorial are contained in the **Information** folder that was created when you installed OS/2.

This manual does not teach the REXX language. Sample REXX code is included in the example programs and a brief introduction to the language is included in the Appendices. You should refer to a REXX manual for more detailed information . Some suggested manuals are listed at the end of this chapter.

## How to use this manual

This manual provides the information you need to create applications with VX- REXX.

Before you read this manual, you should install VX-REXX following the instructions in the 'Setting up' chapter.

## Organization

This manual is divided into four parts:

**Getting Started**

Chapter 1, *Introduction*, provides an overview of the book and other sources of information.

Chapter 2, *Setting up*, tells you how to install VX-REXX

Chapter 3, *A simple program*, gets you started right away by taking you through the creation of a simple VX-REXX program.

**Programmer's Guide**

Chapters 4 to 7, the first part of the Programmer's Guide, contain the information you require to create and manipulate VX-REXX projects. The first chapter, *Creating and running projects*, gives an overview of a project and describes creating, saving, and running projects. The next three chapters, *Adding objects to a window*, *Changing object properties*, and *Adding and modifying routines* explain the mechanics of designing VX-REXX projects.

Chapters 8 to 12 explain how to program with objects. Chapter 8, *Programming with objects*, describes general techniques and functions used with objects. Chapter 9, *Using objects*, presents detailed information on using specific types of objects. Chapter 10, *Bitmaps, icons and resources*, shows how to add bitmaps and icons to your project. Chapter 11, *Drag and drop operations*, shows how to add drag and drop capabilities to your projects. Chapter 12, *Adding menus to a program*, shows how to add menus to a window.

Chapter 13 describes the dialog and system functions provided to simplify application development. The dialog functions allow you to create and process predefined dialogs to display messages, prompt for input, select a font, and select a file . The system functions access file and system services such as file renaming and deleting.

Chapter 14, *Creating custom dialogs*, describes how to create your own custom dialogs by using modal windows.

Chapter 15, *Secondary windows*, explains how to create a user interface where several modeless windows are made available at the same time.

Chapter 16, *Multiple file projects*, explains how to create and manage projects that consist of multiple source files. It also includes information on window and code files, and exchanging information between source files.

Chapter 17, *Adding help to a program*, explains how to add help to your application.

Chapter 18, *Debugging a project*, explains how to debug programs using the VX- REXX interactive debugger.

Chapter 19, *Extending VX¨REXX*, explains how VX-REXX can be extended using object libraries and external function libraries.

Chapter 20, *Database 2 OS/2 with VX¨REXX*, shows how to create database applications with VX-REXX by using the REXX interface to IBM's DB2/2 or Watcom SQL. The techniques shown here to access the database are similar to those you can use to access any package that supports a REXX interface.

Chapter 21, *Writing multithreaded applications*, discusses the way to write multithreaded applications.

Chapter 22, *Controlling other programs*, explains how you can control and communicate with other applications from your VX-REXX project.

Chapter 23, *Creating objects at run time*, explains how objects can be created dynamically at run time.

Chapter 24, *VX¨REXX design time macros*, explains how the VX-REXX design environment can be extended with macros.

Chapter 25, *Distributing your projects*, explains what is needed to distribute your projects.

Chapter 26, *Samples*, summarizes the sample programs that come with VX-REXX .

**Reference**

Chapters 27 to 32 provide a summary of all VX-REXX objects, properties, events, methods, predefined routines, and functions.

In addition, a complete alphabetic reference for all objects, properties, events, methods, predefined routines, and functions is found online in the ***VX¨REXX Reference..***

**Appendices**

Appendix A is a glossary of terms used in this book.

Appendix B is a quick introduction to REXX which you can use either to start learning the REXX language or to refresh your REXX skills.

Appendix C contains a brief summary of the REXX programming language.

## Syntax conventions

Several conventions are used when presenting syntax information.

oBold and normal text indicate words that you must enter as shown:

**say**, VRSet, BackColor, and **Red**.

Unless specifically mentioned, the words must be spelled as shown but can be entered in any combination of upper and lower case letters.

Word constants that are passed to functions must be quoted:

call VRSet object, 'BackColor', 'Red'

oItalic letters indicate placeholders for information you supply:

value = VRInfo( *type* )

You can always use your own name for the variable that receives the return value.

You can always use the **call** instruction to discard return values:

call VRInfo *type*

oOptional arguments are enclosed in square brackets. For example,

fileInfo = VRDir( [ *fileSpec* ], [ *output* ], [ *search* ], [ *mask* ] )

The commas between arguments are not required if all subsequent arguments are omitted:

fileInfo = VRDir( '*.*' )

# Online information

## Online books

All of this manual is provided online. It is divided into two online books: the *VX-REXX Programmer's Guide* and the *VX-REXX Reference*. The programmer's guide contains the chapters from the Getting Started and Programmer 's Guide parts of this manual. The reference contains the chapters from the Reference and Appendices as well as a complete alphabetical reference. You can browse either book by double-clicking on the appropriate icon in the VX-REXX folder.

The online information for VX-REXX is fully hyper-linked. You can also search (with or without wildcards) for text. The design environment includes several features to quickly access online information from various locations.

## REXX information

The VX-REXX folder also contains an icon for the *OS/2 REXX Information*. You can browse this document for information about REXX language statements. This REXX documentation may have been installed when you installed OS/2. If it was not, see Setting up for instructions on adding the REXX Information.

## Context help at design time

Context sensitive help is provided for the VX-REXX design environment. To see help while you are located on a menu or object, press **F1**.

# Technical notes

From time to time, Watcom makes available technical information pertaining to VX-REXX in the form of VX-REXX Tech Notes. Tech Notes are available electronically from the Watcom bulletin board (see the contact information at the front of this book) and on CompuServe (type GO WATCOM). As of this printing, the following Tech Notes are available:

vxtech01.zip How to build external function libraries. Includes sample C source code.

vxtech02.zip (Obsolete) Sending keystrokes to control other programs. The material in this Tech Note has been superseded by the 'Controlling other programs' chapter.

vxtech03.zip How to use REXX queues to communicate with VIO applications (applications that can only run in an OS/2 window or full-screen session).

vxtech04.zip A simple introduction to REXX queues.

vxtech05.zip Customizing the VX-REXX executable stub. Includes sample C source code.

vxtech06.zip Running and controlling other programs from a VX-REXX program. Complements the material in chapter 'Controlling other programs' .

vxtech07.zip How to build REXX-aware applications. Includes sample C source code.

# Other sources of information

Many manuals are commercially available to help you learn about the REXX language. Some of these include:

*Modern Programming Using REXX* by R. P. O'Hara and D. R. Gomberg (ISBN 0-13-579329-5).

*OS/2 REXX: From Bark to Byte* from IBM, document #GG24-4199-00.

*OS/2 REXX Handbook* by Hallette German. (ISBN 0-442-01734-0).

*Practical Usage of REXX* by Anthony S. Rudd (ISBN 0-13-682790-X).

*Procedures Language 2/ REXX Reference V2.00* (S10G-6268) from the IBM OS/2 Technical Library.

*Procedures Language 2/ REXX User's Guide V2.00* (S10G-6269) from the IBM OS/2 Technical Library.

*Programming in REXX* by C. Daney (ISBN 0-07-015305-1).

*The REXX Handbook* by Gabriel Goldberg and Philip H. Smith III (ISBN 0-07- 028682-8).

*REXX -- Advanced Techniques for Programmers* by Peter C. Kiesel. (ISBN 0-07-034600-3).

*The REXX Language* by M. F. Cowlishaw (ISBN 0-13-780651-5).

*REXX Reference Summary Handbook* by Dick Goran. (ISBN 0-9639854-1-8).

*Writing OS/2 REXX Programs* by Ronny Richardson. (ISBN 0-07-052372).

If you want to learn more about OS/2 user interface design you may want to read the IBM Systems Application Architecture manual *Common User Access Advanced Interface Design Reference* (SC34-4290). It provides guidelines for when and how to use the standard interface components.

# Setting up

The setup program for VX-REXX copies VX-REXX from the installation diskettes onto your hard disk. It also creates a Workplace Shell folder with icons for the design environment, the online books, samples, and design time macros.

## Before you start

### Check your VX-REXX package

Check to make sure that your VX-REXX package contains the following materials :

oInstallation diskettes. If you have the VX-REXX Standard Edition there should be 3 diskettes. If you have the Client/Server Edition there should be 5 diskettes.

oThe *VX-REXX for OS/2 Programmer's Guide and Reference*

oThe *VX-REXX Client/Server Edition Objects Guide*. This volume is made up of two books: the *Database Object Guide* and the *Chart Object guide*. These books are included with the Client/Server Edition only.

oYour registration card

### Check your machine

Before installing, make sure your computer fulfills these requirements:

oAn IBM-compatible machine with a 386 (or higher) CPU

oAt least 8 megabytes of memory

oAt least 6 megabytes of hard disk space for the Standard Edition, and an additional 7 megabytes for the Client/Server Edition.

oOS/2 2.0 or later

If you have OS/2 2.0, you must install the Service Pak before installing VX-REXX. You can obtain it from the IBM OS/2 BBS, CompuServe, or directly from IBM. The IBM number for the Service Pak is XR06055. Contact your IBM representative for more information.

oThe OS/2 REXX interpreter

If you are not sure whether the OS/2 REXX interpreter is installed, check your OS2\DLL directory for the following files: REXX.DLL, REXXAPI. DLL, REXXUTIL.DLL, and REXXINIT.DLL. If they are present, the interpreter has been installed.

oThe OS/2 Enhanced Editor (optional)

oThe OS/2 resource compiler (optional)

## Installing OS/2 REXX, the Enhanced Editor and the resource compiler

This section shows you how to install the OS/2 REXX interpreter, the OS/2 Enhanced Editor and the OS/2 resource compiler. If you have already installed these, skip ahead to the next section.

You must install the OS/2 REXX interpreter and the REXX information before installing VX-REXX.

In addition to using the OS/2 REXX interpreter, VX-REXX is also configured to use the OS/2 Enhanced Editor. Although it is not required, you may wish to install the Enhanced Editor to make sure you can use all of VX-REXX's features .

For the following procedure you will need the OS/2 installation diskettes .

1.Open the **OS/2 System** folder.

2.Open the **System Setup** folder.

3.Open the **Selective Install** program.

4.Click on **OK** in the **System Configuration** window.

5.If you need to install the OS/2 REXX interpreter, click on the **REXX** checkbox.

6.If you wish to install the REXX online information, click on the **Documentation** checkbox. Then click on the corresponding **More** button to open the **Documentation** window. Make sure **REXX Information** is checked, then click on **OK**. If other check boxes are checked, you can uncheck them by clicking on their respective check boxes.

7.If you wish to install the OS/2 Enhanced Editor, click on the **Tools and Games** check box. Then click on the corresponding **More** button to open the **Tools and Games** window. Make sure that the **Enhanced Editor** check box is checked, then click on **OK**. If other check boxes are checked, you can uncheck them by clicking on their respective check boxes.

8.Click on **Install** to start the installation. You will be instructed by the **Selective Install** program after this point.

If you wish to bind resources to your projects, you must ensure that the OS/2 resource compiler is installed. It is part of the standard OS/2 installation. To verify that the resource compiler is installed, open an OS/2 window, type **rc** and press **Enter**. If a help screen is displayed, the resource compiler is installed and in the default search path.

You are now ready to install VX-REXX.

# Installing VX-REXX

The first VX-REXX diskette contains a program called **SETUP.EXE** which installs the software.

By default, the software will be installed in the directory called \ VXREXX on the hard disk with the OS/2 system installed, but you can name a different directory. Note the instructions that follow assume that the VX-REXX directory is C :\VXREXX. If you choose another drive or directory, then substitute that directory for C:\VXREXX.

The installation program will ensure that there is enough free disk space before starting the actual installation. If there is insufficient free space, the installer will prompt you to delete some files from your disk and then restart the installer.

The installation program requires a minimum of 1 megabyte of free space on the OS/2 boot drive to start the install.

If you already have a version of VX-REXX installed on your machine you must ensure that it is not in use before installing the new version. You cannot install VX-REXX if you are running either VX-REXX or a VX-REXX program or macro.

*If you are upgrading VX-REXX we strongly recommend that you install the new version of VX-REXX in the same directory as the previous version*. The installer will upgrade all the VX-REXX files but will leave the **Projects** directory and any projects it contains intact.

**Note:** Only one version of VX-REXX can be running at one time. This includes VX-REXX itself and any VX-REXX program or macro. However, all VX-REXX programs and macros created with earlier versions of VX-REXX will run with the current version without change. It is not necessary to regenerate either EXE or macro files if you are upgrading.

To install VX-REXX:

1.Open the **OS/2 System** folder.

2.Open the **Command Prompts** folder.

3.Open an **OS/2 Window**.

4.Insert the first **Watcom VX-REXX** diskette in a diskette drive (e.g . the **A** drive).

5.From the OS/2 command line, type: **a:\setup**.

If you are installing the Client/Server Edition, the installer will ask you to select the components you want to install. You can install the VX-REXX design environment, the Client/Server objects, or both. Check the components you want to install, then press **OK**

The installer will prompt you for the drive and directory in which you want to install VX-REXX. You can change the default values to any valid drive and directory. If the directory does not exist, the installer will create it.

After choosing the VX-REXX directory, the setup program will install the software on your hard disk. From time to time, the installer will prompt you to insert the next disk. When you see this message, remove the old disk, insert the requested disk, then press **OK**.

Once the files have been installed, you will be asked if you want your CONFIG .SYS updated. If you select this option, the installer will update your CONFIG.SYS so that the PATH, LIBPATH, HELP and BOOKSHELF variables include the VX- REXX directory. It will also create a VXREXX variable set to the VX-REXX directory. Your old CONFIG.SYS will be renamed CONFIG.BAK.

If you do not select this option, the installer will create a copy of your CONFIG.SYS with the changes it would have made. It writes this to the CONFIG .VRX file in the C:\VXREXX directory.

VX-REXX will not function properly until you update your CONFIG.SYS. Adding the VX-REXX directory to the PATH allows you to start VX-REXX from an OS/2 command line. The LIBPATH must be updated to run the executable and macro files generated by VX-REXX. The HELP and BOOKSHELF variables must be updated to access the VX-REXX context sensitive help. The VXREXX variable should be set to the VX-REXX directory. Note that you must shutdown and restart OS/2 for changes to your CONFIG.SYS file to take effect.

When the installation is complete, you will be asked if you want to view the **Read Me First**. This document contains important information that was not available when the manual was printed, and is only available online.

After the installer finishes, you will be returned to the OS/2 command line . Type **exit** to close the OS/2 window. At this point you should reboot your computer so that changes to the CONFIG.SYS file can take effect.

## Send in your registration card

You should fill out and mail or fax your registration card as soon as possible to ensure you are informed of future upgrades and other special offers to registered users. You must write the product registration number on this card in the appropriate box. Your registration number is printed on the label of the first diskette . You will need your registration number if you contact Watcom Technical Support .

## Watcom VX-REXX folder contents (Standard Edition)

The installer creates a Watcom VX-REXX icon on your desktop. Open this folder to view the VX-REXX contents. This section describes each of the icons the folder contains.

Note that the Watcom VX-REXX folder on your desktop only contains objects that reference files in the VX-REXX directory. Deleting or moving this folder does not delete or move the VX-REXX directory. If you accidentally delete the Watcom VX-REXX folder, you can rebuild it with the **BUILDVRX** command which is described later.

### VX-REXX

You can use the **VX-REXX** icon to open the VX-REXX development environment, although usually you will open VX-REXX by opening a **VX-REXX Project** object. See the section on the **Projects** folder in this document for more information.

## Read Me First

The **Read Me First** document contains information that was not available when the *Watcom VX-REXX for OS/2 Programmer's Guide and Reference* went to press . Look in this document for corrections to the manual and documentation for things not covered in the printed manual.

## VX-REXX Programmer's Guide

**VX-REXX Programmer's Guide** is an online copy of the **Programmer's Guide** section of the **Watcom VX-REXX for OS/2 Programmer's Guide and Reference** . You can read this documentation to learn how to use VX-REXX and create VX-REXX applications. You can also use the chapter 'A simple program' as an online tutorial, actually building the program in the VX-REXX development environment as you read through the chapter.

You should work through the tutorial before trying to create your own project . It shows you how to examine the code and how to run a VX-REXX program. If you need more information, the techniques outlined in the tutorial are expanded in the first four chapters of the **Programmer's Guide**.

## VX-REXX Reference

The online **VX-REXX Reference** is a full alphabetical reference which lists all objects, properties, methods, and functions in the Standard Edition of VX-REXX. Use this documentation to get detailed information on any component of the VX-REXX development system.

## REXX Information

**REXX Information** points to the online REXX language documentation which is contained in the OS/2 **Information** folder. You can browse this information to learn about the REXX language and the OS/2 **REXX** interpreter.

## Samples

This folder contains icons for the VX-REXX sample programs. The source for the samples is in the **Samples** folder. The samples are described in the 'Samples' chapter.

To run these sample programs, the LIBPATH and HELP environment variables must include the VX-REXX directory.

### Projects

The **Projects** folder contains a shadow of the **VX-REXX Project** template. You use this template to create new VX-REXX projects. Simply drag the template to the folder in which you want to store the new project files. Open the **VX- REXX Project** folder that is created, and open the **Project.VRP** file to start VX- REXX with the new project.

The **Projects** folder is a shadow of the directory of the same name in the VX- REXX directory. You can use this folder to safely store your projects as they will remain intact even if the Watcom VX-REXX folder is accidentally deleted.

### Macros

The **Macros** folder contains sample VX-REXX design time macros, and information on developing your own macros. VX-REXX design time macros allow you to customize the design environment by adding new commands to the VX-REXX pop-up menu.

Three sample macros are provided: **SetProps** (another way to set properties), **Resize** (force objects to the same size), and **ObjList** (list all objects on a window). The source for these macros is included so you can modify and adapt them to suit your own needs.

Information on the sample macros and their installation instructions is included in the **Profile.VRM** file located in the **Macros** folder. Details on writing your own macros are contained in a later chapter in this book.

The **Macros** folder is a shadow of the directory of the same name in the VX- REXX directory. You can use this folder to safely store your macros as they will remain intact even if the Watcom VX-REXX folder is accidentally deleted.

### Color Palette

The VX-REXX installation program creates a color palette Workplace Shell object in the VX-REXX folder. You can use the palette to drag and drop colors onto VX-REXX objects as an alternative to setting colors using a property notebook.

### Font Palette

The VX-REXX installation program creates a font palette Workplace Shell object in the VX-REXX folder. You can use the palette to drag and drop fonts onto VX-REXX objects as an alternative to setting fonts using a property notebook.

## Watcom VX-REXX folder contents (Client/Server Edition)

### Database Administrator

Double click on the **Database Administrator** icon to start the database administrator tool.

### Chart samples

The **Chart samples** folder contains a number of samples that demonstrate the chart object.

### Database samples

The **Chart samples** folder contains a number of samples that demonstrate the database objects.

### Chart Object Guide

This book is the programmer's guide and reference for the chart object . It is an online copy of the printed *Chart Object Guide*. Corrections to the printed book listed in the **Read Me First** have been made to the online version.

### Database Objects Guide

This book is the programmer's guide and reference for the database objects. It is an online copy of the printed *Database Objects Guide*. Corrections to the printed book listed in the **Read Me First** have been made to the online version.

## Watcom VX-REXX directory contents

In addition to the files and directories comprising the VX-REXX design environment, the VX-REXX directory contains the following useful files.

### Buildvrx.CMD

This REXX script rebuilds the Workplace Shell objects for VX-REXX. If you accidentally delete your **Watcom VX-REXX** folder, or if your desktop becomes corrupted, you can run this command file from an OS/2 window.

### Buildcso.CMD (Client/Server Edition only)

This REXX script rebuilds the Workplace Shell objects for the Client/Server Edition. If you accidentally delete your **Watcom VX-REXX** folder, or if your desktop becomes corrupted, you can run this command file from an OS/2 window.

## Bpatch.EXE

The Watcom Patch Utility can be used to apply patches or bug fixes to VX-REXX . As problems are reported and fixed, patches are released on the Watcom BBS and FTP site. You can also order the patches on diskette from Watcom.

## Recover.EXE

If your computer crashes while running VX-REXX, you may be able to recover your work using the RECOVER program. To run it, follow these steps:

1.Open an OS/2 window

2.Change to your VXREXX directory. Type **cd c:\vxrexx** ( Replace 'c:' with the actual drive letter).

3.Start the RECOVER program by typing **RECOVER**.

4.Select the directory which contains the source for your project.

5.Press the **Recover** button.

For detailed help on the recovery process, press **F1** while running the RECOVER program.

# A simple program

This chapter provides a brief introduction to the VX-REXX development environment through a tutorial. You will learn how to start and stop VX-REXX and create , run, and save a simple application. In addition, the drag and drop programming feature of VX-REXX will be used to reduce the time and effort needed for writing code. The same steps used in this tutorial can be used to develop all VX- REXX applications. Subsequent chapters provide more details about each step.

In this tutorial, you will create an application which consists of an entry field, push button, and list box. When run, text typed in the entry field will be added to the list box using the push button.

## Starting VX-REXX

You must install VX-REXX before working through this tutorial. During the installation the **Watcom VX-REXX** folder is created on the OS/2 desktop. Open that folder now. You will see several icons and a number of folders.

To start a new VX-REXX project:

1.Open the **Projects** folder.

2.Drag the project template to a blank area in the **Projects** folder. A project folder will be created.

3.Open the folder. Three icons will be displayed. They represent the files in the new empty project.

4.Double click on the **Project.VRP** icon to start VX-REXX.

When VX-REXX has started, the screen will appear as in Figure 1:

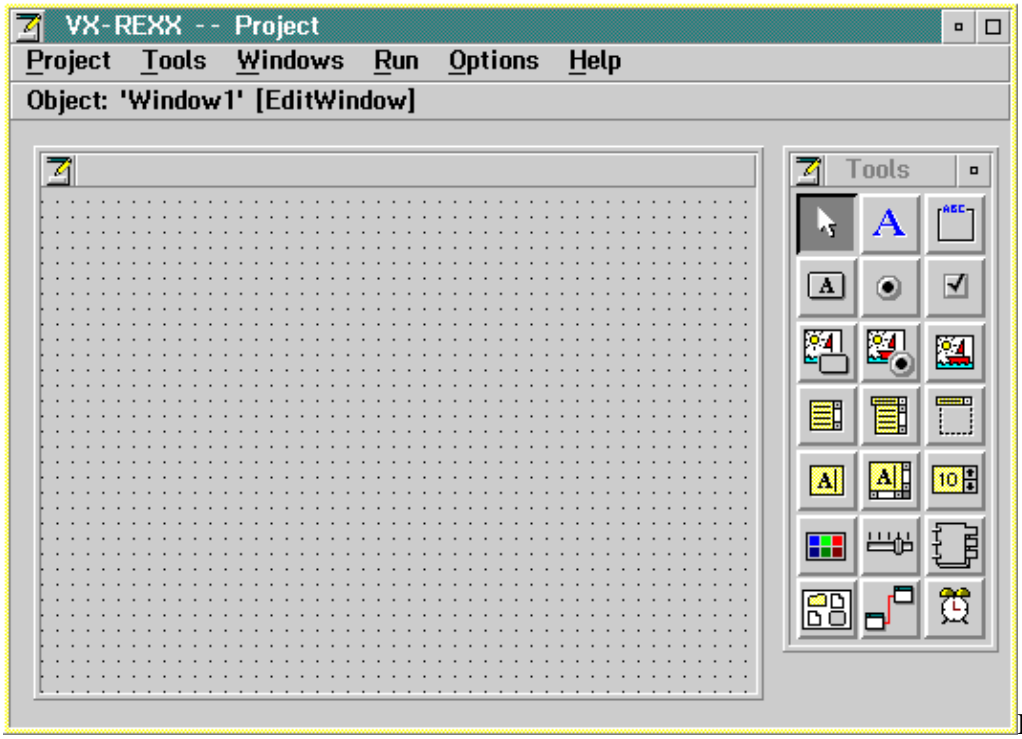


**Figure 1** The initial VX-REXX screen.

The VX-REXX design environment consists of a menu bar, a tool palette, and a project window containing a grid of dots. The menu bar contains the commands that are used to create, save and run applications. The tool palette contains the objects which can be placed in the project window to create the user interface for an application.

## Creating an application

There are three basic steps to building a VX-REXX application:

1.Creating the user interface by adding objects to the project window .

2.Customizing each object's properties.

3.Attaching code to each object.

You will go through these steps in the following sections to create a simple application.

## Creating the user interface

Creating the user interface is the first step in building an application. The interface to this application has three objects: an entry field, a push button, and a list box. Start by adding the entry field to the window:

1.Click on the **Tools** menu and then choose **EntryField**.

2.Position the mouse pointer near the top left corner of the project window.

The cursor will change to a cross while it is positioned on the project window.

3.Hold down mouse button 1 and drag the mouse down and to the right until the entry field is approximately the size of the one shown in Figure 2.

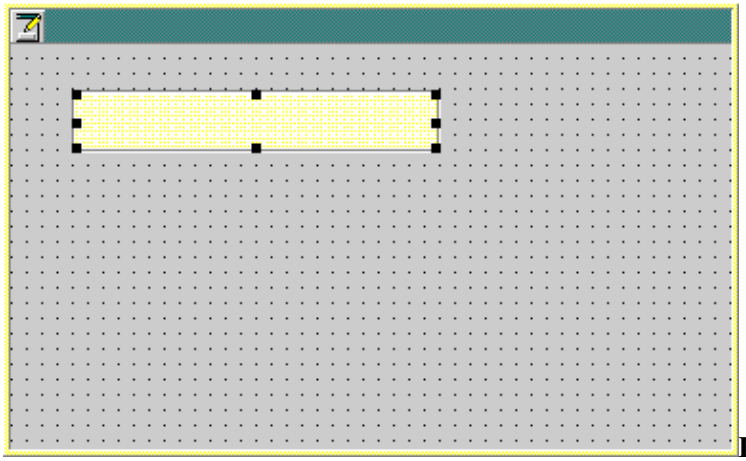

**Figure 2** The entry field added to the window.

4.Release the mouse button.

Add the push button to the window using a similar procedure:

1.Click on the **Tools** menu and then choose **PushButton**.

2.Position the pointer below the bottom left corner of the entry field .

3.Hold down mouse button 1 and drag the mouse down and to the right until the push button is approximately the size of the one shown in Figure 3.

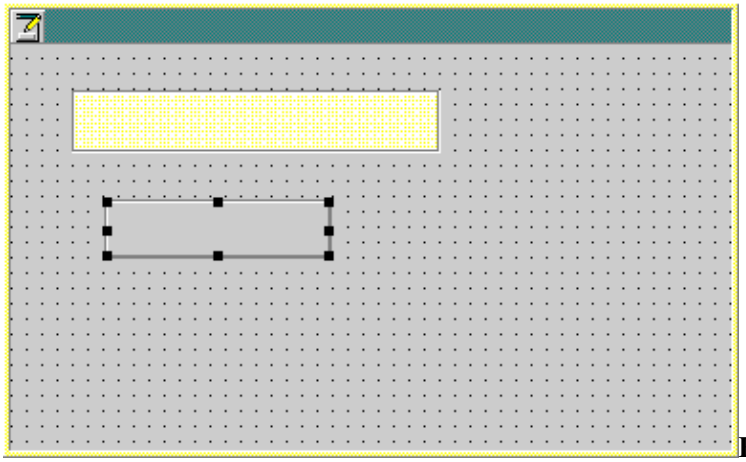

**Figure 3** The push button added to the window.

4.Release the mouse button.

Repeat the above steps using the **ListBox** item from the **Tools** menu to make your window similar in appearance to Figure 4.

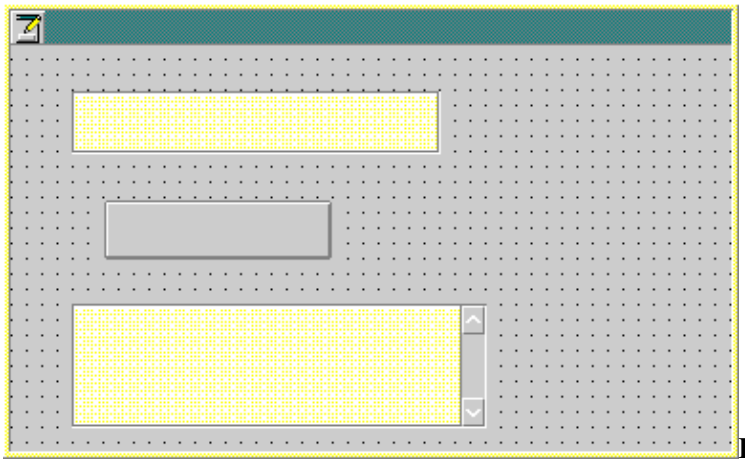

**Figure 4** The entry field, push button and list box on the window.

Finally, to improve the appearance of the window, center the three objects on the window as follows:

1.Position the pointer over the entry field.

2.Press and hold down mouse button 1. Sizing handles will appear along the border of the entry field. Move the pointer over the push button and then over the list box. Now all three objects will have sizing handles along their borders. The screen should look similar to Figure 5.

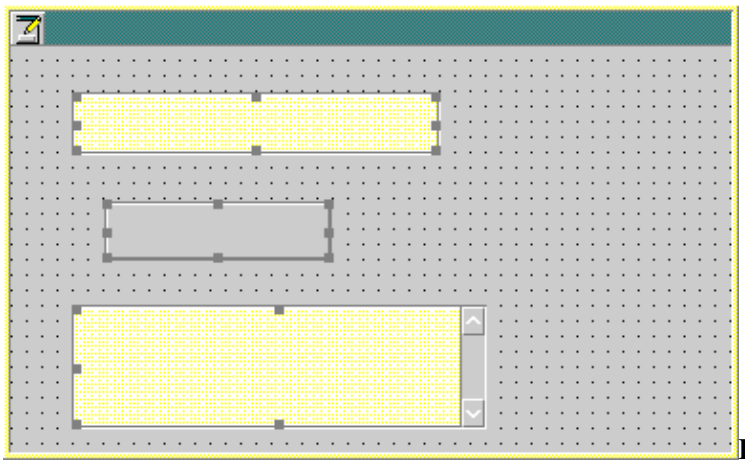

**Figure 5** The entry field, push button and list box are selected.

3.Release mouse button 1.

4.Position the pointer on the entry field once again and press mouse button 2 to invoke the VX-REXX pop-up menu.

5.Select the **Align** menu item using mouse button 1. A second menu will appear. From this menu, click on the menu item used for centering objects as shown in Figure 6.

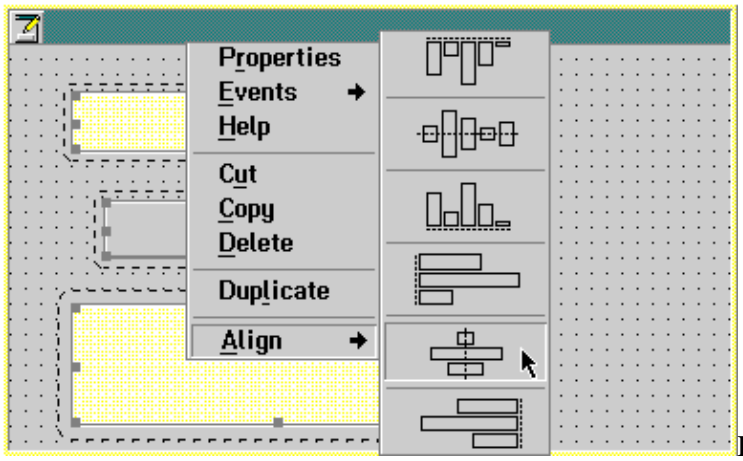


**Figure 6** The menu item used to centre objects.

The push button and the list box will now be centered underneath the entry field .

6.Move the pointer over the right border of the window. The pointer will change to a sizing arrow.

7.Hold down mouse button 1 and move the cursor left or right to adjust the width of the window until the objects are centered.

8.Click on the window background to deselect the objects.

The screen should appear as in Figure 7.

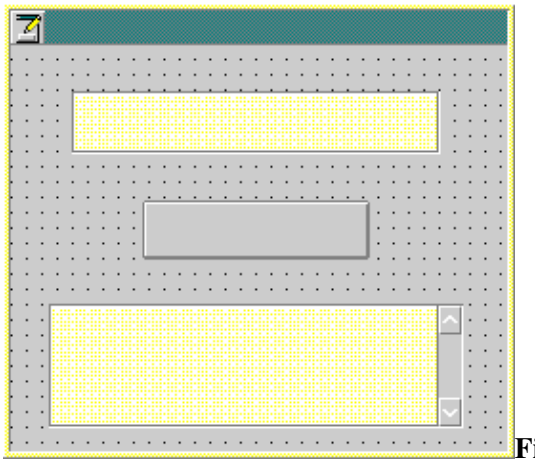

**Figure 7** The objects centred on the window.

All the objects you need have been created. Now their properties must be customized.

# Customizing object properties

Each VX-REXX object has properties such as color and size which can be customized. An object is given default property values when it is first created. Often, these properties need to be changed.

The entry field has a font property which determines the font used to display text. The height of the entry field should be set according to the size of the font being used. This step can be done automatically using the entry field' s Autosize property. If this property is set to 1, the height of the entry field will be adjusted automatically based on the font being used. Initially, the Autosize property is set to 0. Change it to 1 as follows:

1.Click mouse button 2 anywhere over the entry field. A pop-up menu will be displayed.

2.Choose the **Open** menu and then choose the **Properties** menu item. A property notebook will be displayed.

3.Click on the **Size** index tab. The **Size** page will be displayed as in Figure 8.

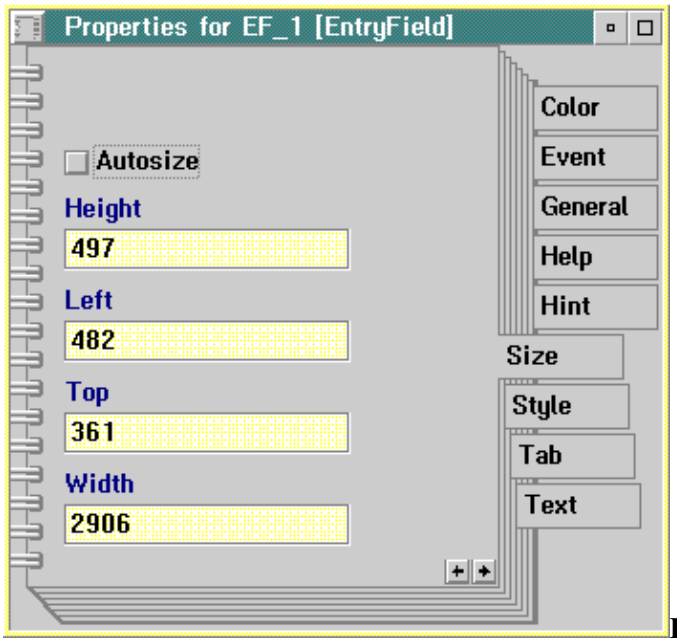


**Figure 8** The size page of the property notebook.

4.Click on the check box marked **Autosize**.

5.Close the entry field's property notebook by double-clicking on the system menu in the title bar.

The text which appears on a push button is called the caption and is determined by the push button's Caption property. To set the caption of the push button to **Add Item**:

1.Click mouse button 2 anywhere over the push button, choose **Open** and then choose **Properties** to open the push button's property notebook.

2.Click on the **Text** index tab. The **Text** page will be displayed as in Figure 9.

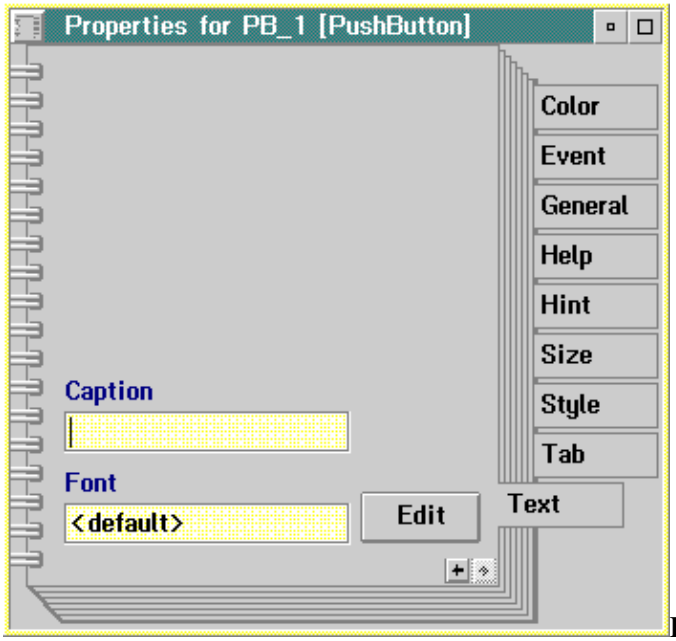


**Figure 9** The text page of the property notebook.

3.Click on the **Caption** entry field and type: **Add Item**

4.Close the property notebook. The text **Add Item** will appear on the push button.

Lines in a list box are displayed one below another, beginning at the top . The last line in the list box may not appear completely, depending on the height of the list box. The height must be adjusted if this characteristic is undesirable. To have the height adjusted automatically, set the AdjustHeight property of the list box as follows:

1.Open the property notebook for the list box using the same method as before.

2.Turn to the **Size** page by clicking on the **Size** index tab.

3.Click on the **AdjustHeight** check box.

By default, items in a list box will be sorted in ascending order. For this application, you want items to be listed in the order they are added to the list box. The list box's **Sort** property must be changed:

1.With the list box property notebook open, turn to the **List** page by clicking on the **List** index tab.

2.Click on the scroll arrows in the **Sort** drop down combobox.

3.Choose **none** from the drop down list.

4.Close the property notebook by double-clicking on its system menu.

Finally, set the text that will appear on the window title bar:

1.Open the property notebook for the window. Note that the pointer must be positioned on the window but not over any object in the window when pressing mouse button 2.

2.Turn to the **Text** page of the notebook. If an index tab for the **Text** page is not immediately visible, click on the arrow below the other index tabs to scroll through the page list.

3.Click on the **Caption** entry field and type:

```
Sample
```

```
4.Close the property notebook.

The first two steps in developing the sample application are complete:
creating the objects and customizing their properties.
```

## A preliminary run

Although the sample program is not finished you can run it to see what it will look like when complete. To run it, click on the **Run** menu and choose **Run project**. This will hide VX-REXX and run the application, which should appear similar to Figure 10:

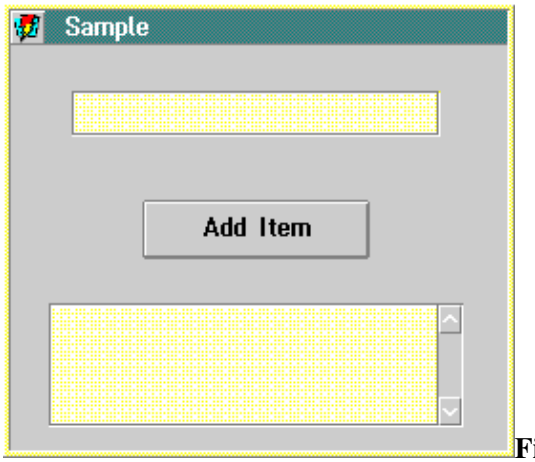


**Figure 10** A test run of the sample program.

Notice that the background of the window no longer contains a grid of dots . Type text into the entry field to see how it will appear. Use the **Tab** key to move between the objects. Click on the push button. Nothing happens because no code has been attached to the push button. Attaching code to the objects in the window is the next step in developing the sample application.

Close the application by clicking the system menu and choosing **Close**. The VX-REXX design environment will reappear.

## Attaching code to objects

In this application pressing the push button will copy the text from the entry field to the list box. You will use the drag and drop programming feature of VX-REXX to help write the code which will perform this task.

First you must obtain the text in the entry field. To attach this code to the push button:

1.Open the push button's property notebook and turn to the **Event** page. Code can be attached to the push button for any events which appear on this page.

2.Select **Click** and then click on **Open** to invoke an editor. The editor will contain a skeleton for an event routine, as shown in Figure 11.

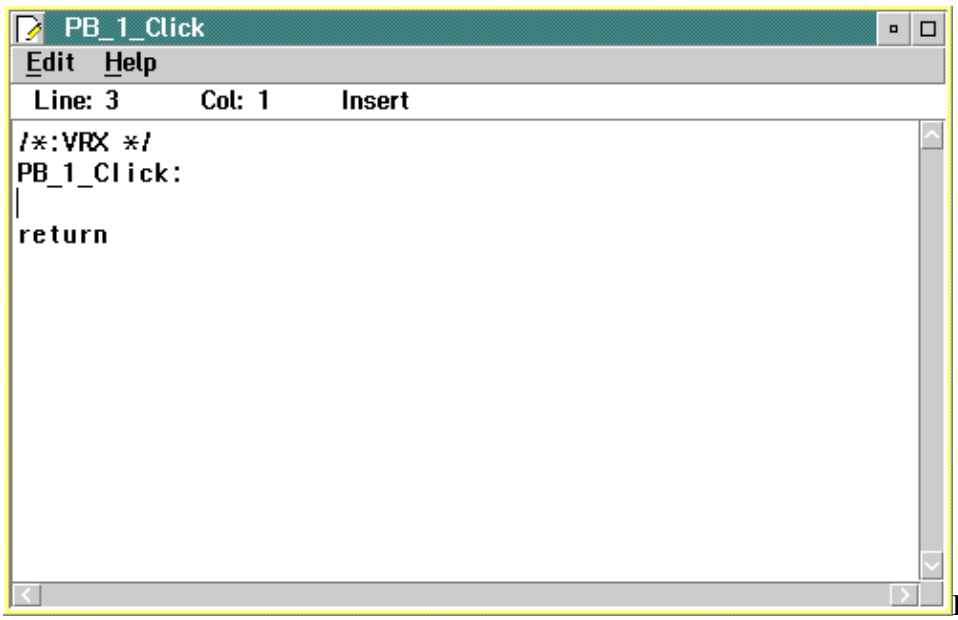


**Figure 11** Event routine skeleton

The first line in the routine is a comment, the second is the name of the routine, and the last line contains a REXX **return** instruction which needs to be executed at the end of the routine. The REXX code you place in this routine will be executed when the push button is pressed.

3.At this point, you could write code manually but there is a much easier method. Press and hold down mouse button 2 over the entry field. Now move the mouse over the editor. The editor window and the application window may need to be moved apart before this step can be done. A line joining the mouse to the entry field will appear as in Figure 12:

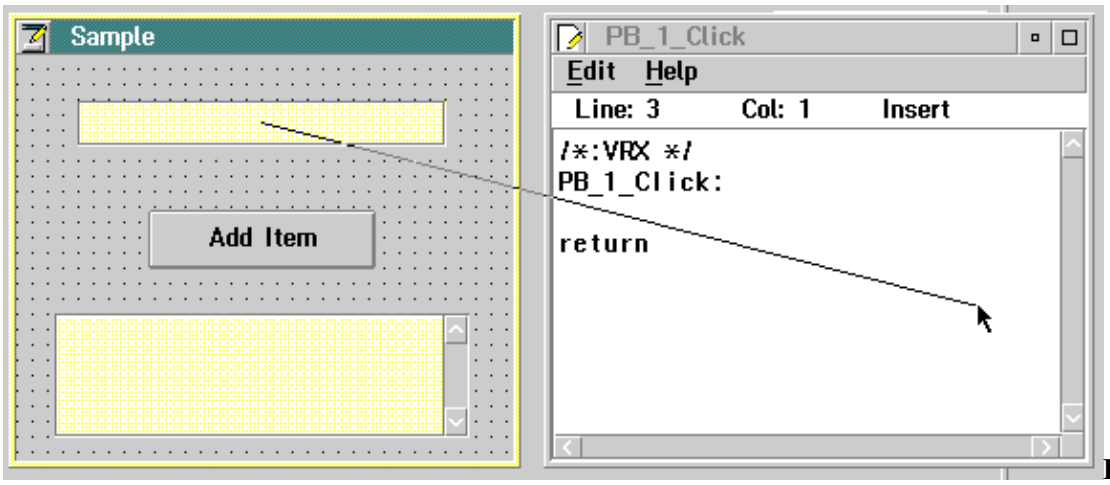


**Figure 12** Getting the text in the entry field.

4.Release mouse button 2. A dialog will appear which lists descriptions of all the entry field methods which can be invoked and all the entry field properties which can be set or retrieved.

5.Click on the **Value** item under the **Get property** heading. You need to get the **Value** property because it contains the text in the entry field. Now click on **OK**. The following instruction is placed at the insertion point in the editor:

```
value = VRGet( 'EF_1', 'Value' )
```

This instruction will assign the variable **value** with the text in the entry field.

The automatic code generation feature of VX-REXX is called drag and drop programming. Using this feature, you can create REXX applications without having to memorize the syntax of VX-REXX functions and methods.

To see the online information for VRGet, hold down the **Ctrl** key while double clicking the mouse on the word. This technique may be used to view online information for any VX-REXX function, event, property or method.

The next step is to add the text to the list box:

1.Press and hold down mouse button 2 over the list box and then move the pointer over the editor window. A line joining the pointer to the list box will appear.

2.Release mouse button 2. A dialog will appear listing all the list box methods that can be invoked and all the list box properties which can be set or retrieved. Click on the **Add a string** item under the **Methods** heading and then press the **OK** button.

3.Another dialog, as shown in Figure 13, will appear.

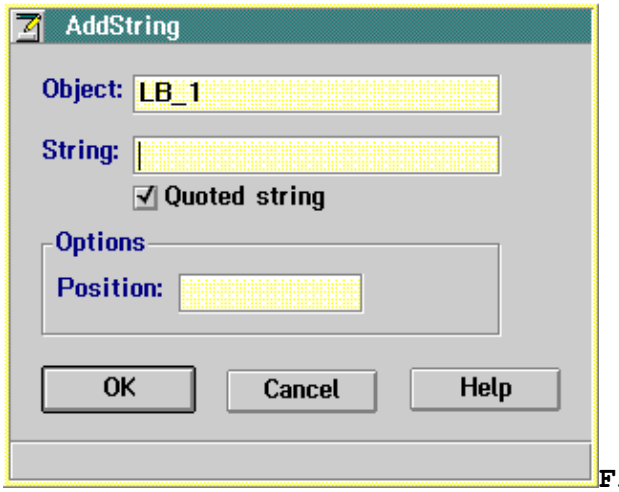


**Figure 13** The Add a String dialog box.

4.Click on the string field and type the following:

```
value
```

5.Click on the **Quoted string** check box. Now it should not be checked .

6.Press the **OK** button. The REXX code to add the text to the list box will be placed underneath the previous instruction, as shown in Figure 14.

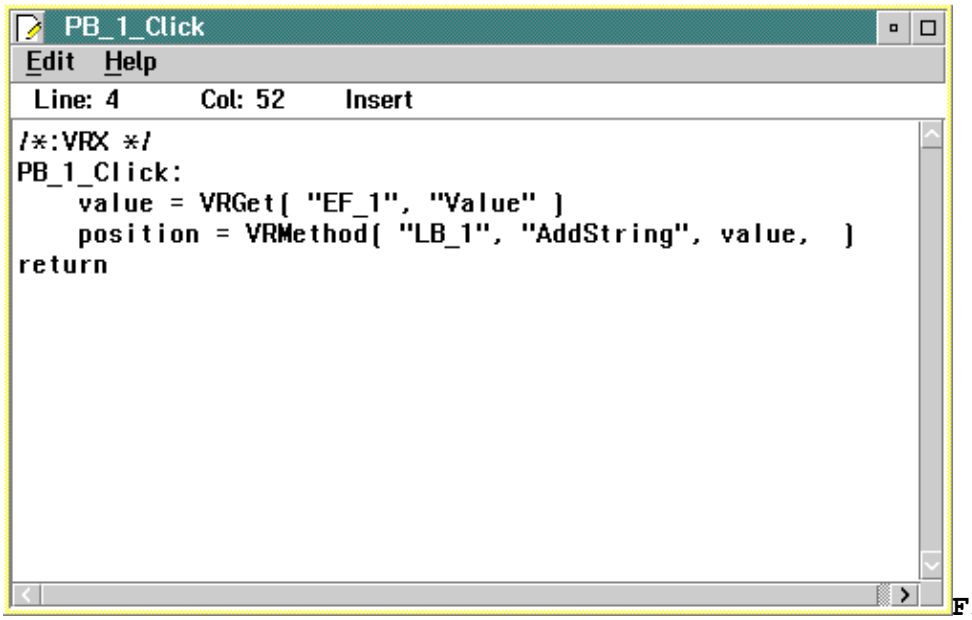


**Figure 14** The event routine for the push button.

After getting the text, you should clear the entry field to allow new text to be entered:

1.Press and hold down mouse button 2 over the entry field and then move the mouse over the editor window.

2.Release mouse button 2. A dialog will appear.

3.Click on the **Value** item under the **Set property** heading and then press the **OK** button.

4.Click on **OK** from the second dialog. The editor window should now appear as in Figure 15.

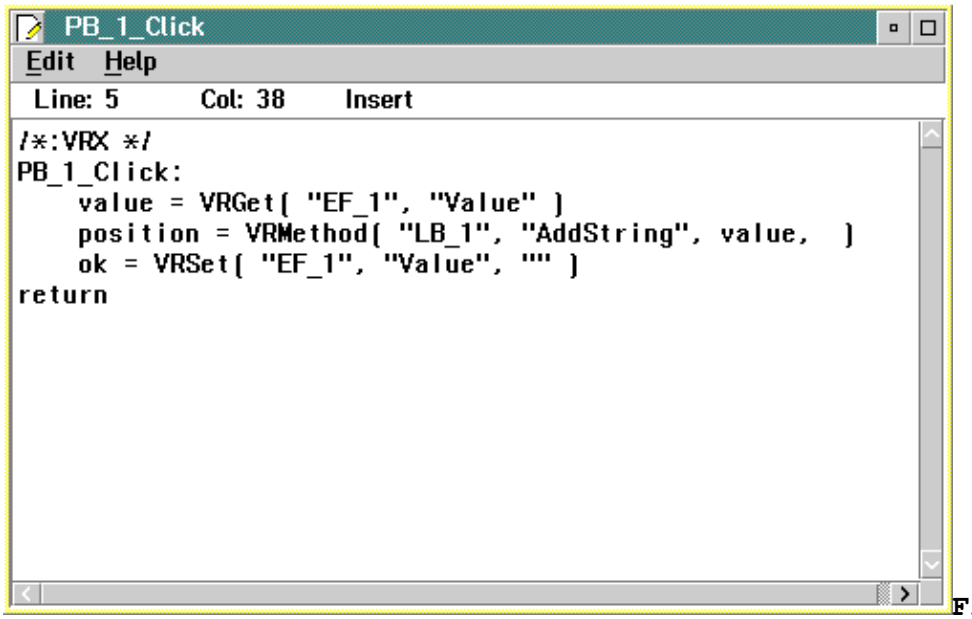


**Figure 15** The unfinished push button event routine.

The push button will have the focus when this routine is executed. You should set the focus to the entry field so that new text can be entered immediately :

1.Press and hold down mouse button 2 over the entry field and then move the mouse over the editor window.

2.Release mouse button 2.

3.Click on the **Set focus to object** item underneath the **Methods** heading and then press the **OK** button. A line of code will be inserted below the previous lines in the editor window.

4.Close the editor window by double-clicking on its system menu icon.

The push button event routine is now complete and is shown in Figure 16. Give the project a test run to see if it works correctly.

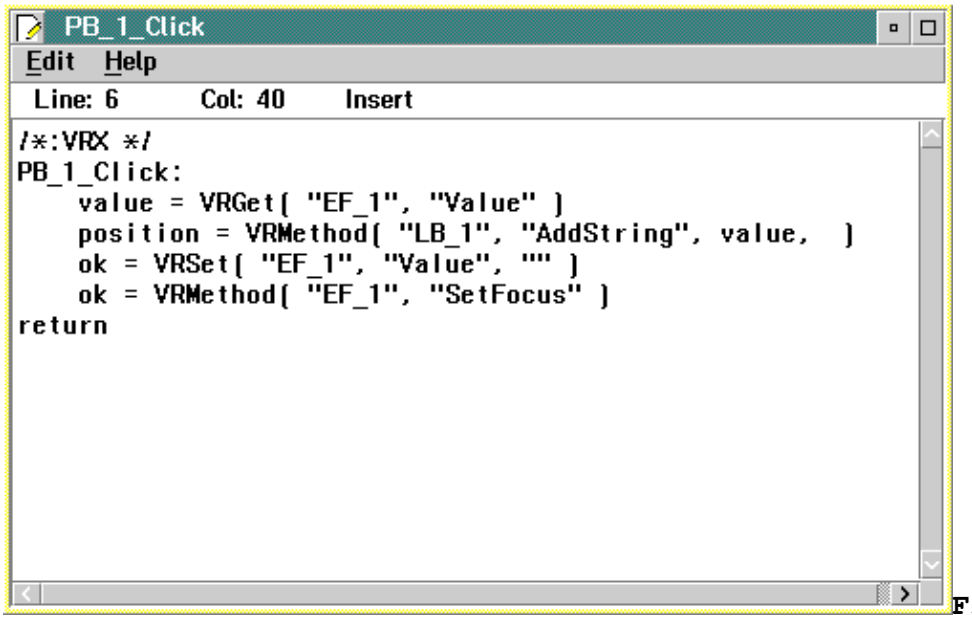


**Figure 16** The complete push button event routine.

## A test run

To run the application, click on the **Run** menu and choose **Run project**. The application window will look the same as it did during the first run. However, try clicking on the push button after typing text into the entry field. Now the text is added to the list box. Also, the entry field is cleared and is given the focus again. The push button click routine you created appears to be working as it should.

Try adding more text to the list box. Also, try pressing the push button when there is no text in the entry field. It seems as though no action is performed. However, if more text is added to the list box, you can see that a blank line is inserted when the entry field is empty and the push button is pressed, as shown in Figure 17.

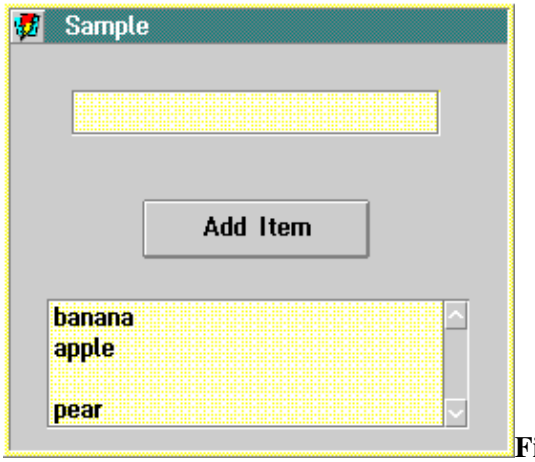


**Figure 17** Adding text to the list box.

This behavior is undesirable and needs to be prevented with the addition of more code. Also, to make the application easier to use, you should allow text to be added to the list box with the use of the **Enter** key.

## Improving the application

A push button can be enabled or disabled. When enabled, a push button responds to events such as a click event. The push button in our application has always been enabled. When a push button is disabled it does not respond to any events and also appears grayed. In particular, a disabled push button cannot receive the focus and will not respond when pressed. The program will use this property of push buttons to prevent blank lines from being added to the list box:

1.Press mouse button 2 anywhere over the entry field to open the VX-REXX pop-up menu.

2.Click on **Events** and then click on **Change**. The editor window will appear with the skeleton of a routine, as shown in Figure 18.

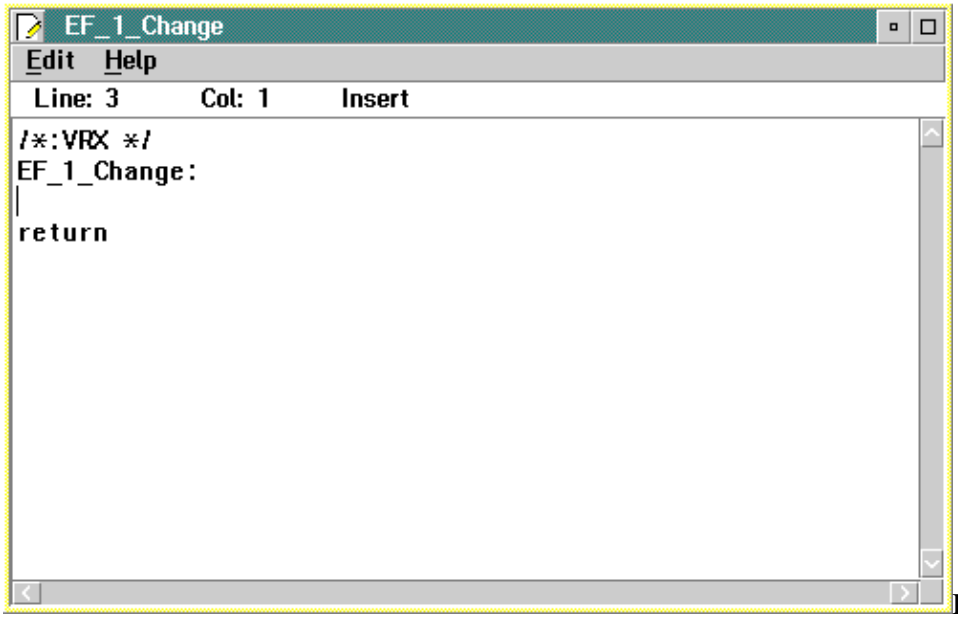


**Figure 18** The entry field change routine.

This routine will be executed whenever the contents of the entry field change . It needs to determine if the entry field has become empty and if this is the case, will disable the push button to prevent it from adding any text to the list box.

3.Press and hold down mouse button 2 over the entry field, move the mouse over the editor window and then release the mouse button. As before, a dialog will appear.

4.Click on the **Value** item underneath the **Get property** heading and then press **OK**. The code to get the contents of the entry field will be generated.

5.Add the following line of code at the insertion point:

```
if value = '' then do
```

```
Press the Enter key and then the Tab key at the end of this line.
```

```
6.At this point, the code has determined that the entry field is empty so
it must disable the push button. Press and hold down mouse button 2 over
the push button, move the mouse over the editor window and then release
the mouse button . A dialog listing all methods and properties of a push
button will appear.
```

```
7.Click on the Enabled item under the Set property heading and then press
OK. A second dialog will appear, as shown in Figure 19.
```

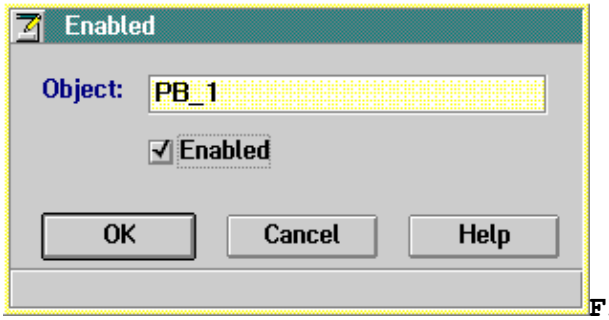


**Figure 19** The enabled property dialog box.

Click on the **Enabled** check box so that it is not checked. Press **OK**. The instruction to disable the push button is placed at the insertion point in the editor window.

8.Now you have to add the code that will enable the push button when the entry field is not empty. Add the following lines at the insertion point:

```
end
else do
```

Press **Enter** and then **Tab** at the end of the second line. The editor window should appear as in Figure 20.

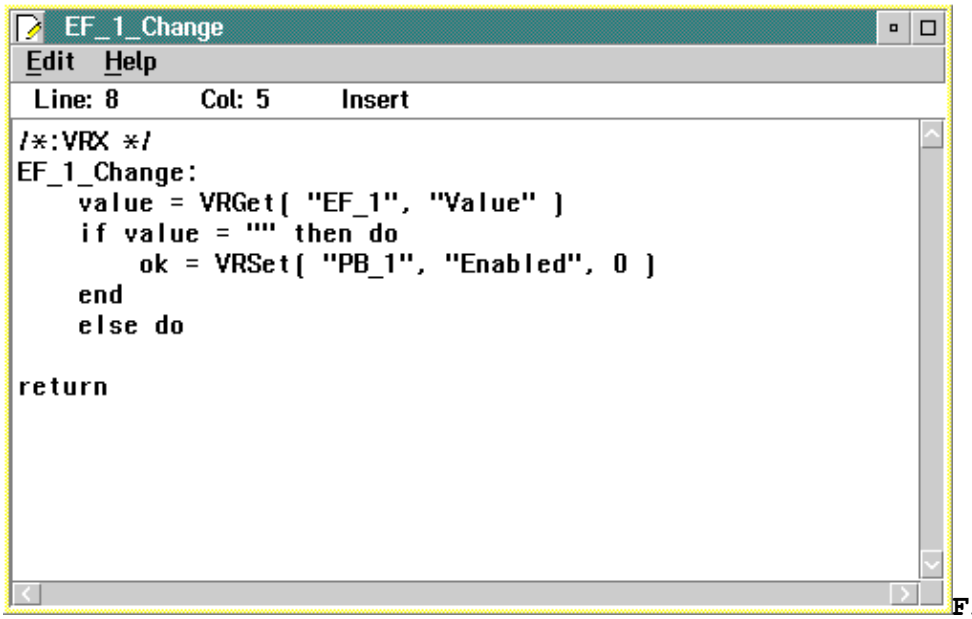


**Figure 20** The incomplete entry field change routine
.

Press and hold down mouse button 2 over the push button, move the mouse

over the editor window and then release the mouse button. From the dialog box click on the **Enabled** item under the **Set property** heading and then press **OK**. Press **OK** again on the second dialog box which appears. The code to enable the push button is inserted into the editor window.

9.One last line needs to be added to the routine:

```
end
```

The complete routine is shown in Figure 21.

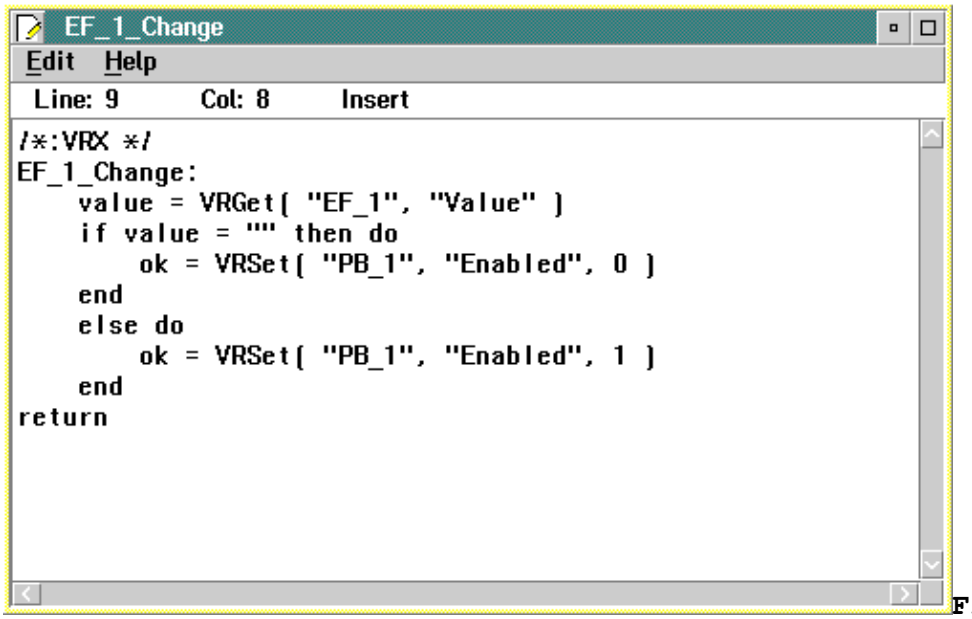


**Figure 21** The complete entry field change routine.

Now you must make the push button disabled when the application is first run :

1.Open the push button's property notebook and turn to the **Style** page. Click on the **Enabled** check box so that it is not checked.

2.Also on this page is the **Default** check box. Click on this check box so that it becomes checked. Now, when the **Enter** key is pressed the push button click routine will be executed, as though the push button had been pressed. This behavior will allow text to be added to the list box by pressing the **Enter** key.

You can easily make another improvement to the application by adding hints to the objects:

1.Turn to the **Hint** page of the push button's property notebook. Type the following in the **Hint** field:

Add the text to the list box.

2.Open the entry field's property notebook, turn to the **Hint** page and type the following:

Enter text.

3.Open the window's property notebook, turn to the **Hint** page and click on the **ShowHints** check box. Set the **Status Area** field. to **Bottom**. A status bar will appear at the bottom of the window, as shown in Figure 22.

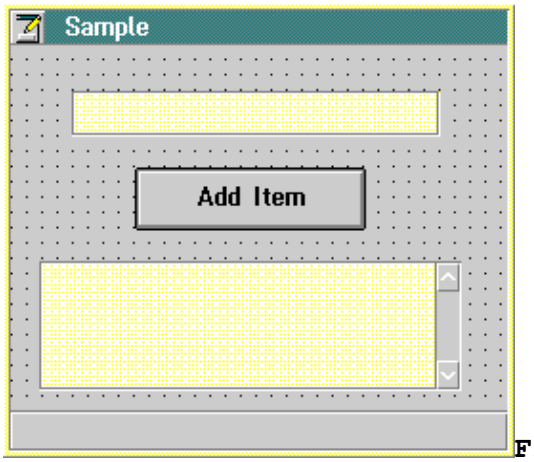


**Figure 22** The status bar for showing hints.

The status bar may overlap with the bottom of the list box. If this is the case then the list box can be raised or its height can be decreased.

The previous steps add hints to the application. When the mouse is moved over the push button, its hint text will be displayed in the status bar. Similarly, when the mouse is positioned over the entry field, its hint text will be displayed. You can now try out the improvements you have made.

## Running the application

Run the application as before by clicking on the **Run** menu and choosing **Run project**. Move the mouse over the entry field and then the push button to see their hints. The application window is shown in Figure 23.

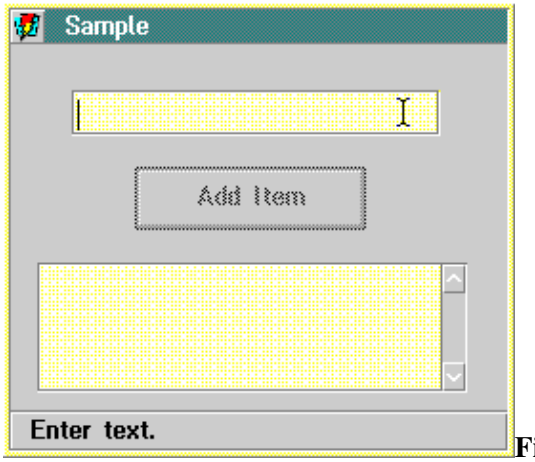


**Figure 23** Running the finished application.

Try typing in text and adding it to the list box with the use of the **Enter** key as well as the push button. Notice that the push button becomes disabled whenever the entry field is empty.

Although this is a simple application, it would take considerably more effort and expertise to create it with other languages and development tools. However, with VX-REXX features like as drag and drop programming, you have created it easily with little knowledge of OS/2 or the REXX language.

## Saving the application

You should save the application now that it is complete:

1.Click on the **Project** menu.

2.Choose **Save**. The project will be saved in its project folder.

## Stopping VX-REXX

To exit VX-REXX and return to the OS/2 desktop, click on the VX-REXX system menu icon and choose **Close** or double-click on the system menu icon.

In this tutorial, you have seen the following techniques:

1.Adding objects to a window.

2.Modifying an object's properties.

3.Using the VX-REXX drag and drop programming feature to add code.

4.Testing a project.

5.Saving a project.

These are the basics. The following chapters expand on these basics to show how to create and maintain complex as well as simple applications.

# Creating and running projects

VX-REXX is a complete REXX programming system, not just a set of libraries that adds features to the REXX language. Its project management facilities, described in this chapter, allow you to easily design and maintain applications.

There are three basic steps to building a VX-REXX application. Building the user interface is the first step. You create the interface by placing control objects such as push buttons in a window. Once they are in place you can easily move and resize them until you are satisfied with their position and appearance . When you have finished designing your window, you have completed the first step and have a prototype for your application. The next chapter shows you how to add objects to a window.

The second step is to assign properties to the objects. The properties of an object determine both how it appears and how it functions. When you create an object, it will have default values for its properties. The chapter *Changing object properties* shows you how to change the default values.

The third step attaches code to the objects to provide action for your application. As well as properties, each object has a set of events to which you can attach REXX routines. An event routine is run when you trigger the event, for example, by clicking on a button. The chapter *Adding and modifying routines* shows you how to do this.

## Projects

A VX-REXX project is a set of one or more window or code files that are combined to form a standalone application. You work on a single project at a time, designing the windows and attaching REXX code to the objects on the windows, testing and debugging the project as necessary. You can generate a standalone executable file at any point in the process. You can run an executable on any machine that has the VX-REXX run time libraries installed.

It is important to distinguish between design mode and run mode. You are in design mode when you are working on a VX-REXX project. You are in run mode when you are testing or debugging a project, or when running a standalone executable . A project can only be modified at design time, but new objects can be created and object properties modified at run time. Some objects also have properties that are only available at run time.

There are several files associated with a VX-REXX project. We recommend that you keep all the files for one project together in a single directory or folder . This will happen automatically if you create your projects by dragging and dropping the VX-REXX project template.

Each project consists of a project file (extension VRP) and one or more source files (extension VRX). Some VRX files require an additional information file. These files have the same name as the VRX file but end with the extension VRY.

Generally you will not need to worry about the details of the files in a project as VX-REXX will maintain them for you. You must be careful, however, when copying or backing up a project to copy or backup all required files.

# VX-REXX and the Workplace Shell

When VX-REXX is installed, a VX-REXX folder is created on the desktop. This folder contains several icons and two folders: Samples and Projects. You can start VX-REXX simply by double clicking on the VX-REXX icon, but the preferred method is to drag and drop a project template. This procedure is outlined in the next section which describes how to create a project.

When installed, VX-REXX also creates Workplace Shell associations between projects and VX-REXX project files. In particular, files with the extension VRP are assumed to be VX-REXX projects. The Workplace Shell will start VX-REXX when such a file is opened. Projects may also be opened by dragging the project file (VRP) onto the VX-REXX program icon.

If you accidentally delete the VX-REXX folder, or must reinstall the OS/2 desktop, the online *Read Me First* contains instructions on recreating the VX-REXX Workplace Shell objects.

## Creating a project

There are two ways to start a new project. The best method is to open the VX-REXX Projects folder and drag and drop the project template contained within it . A new project folder will be created. Open the folder and double click on **Project.VRP** to edit the new project.

Be careful when dragging a project template. It is best to use a folder on one of your Drives objects as the target. This ensures that the project is not lost if your desktop must be reinstalled. The VX-REXX folder in particular is not a good choice, since its contents will be replaced completely if a new version of VX-REXX is installed. The Projects folder inside the VX-REXX folder is safe, however, because it is a shadow of a physical directory on the drive where you installed VX-REXX. Once you have created a project in a safe location, you may of course place a shadow of the project folder wherever you wish.

You can also create projects from within VX-REXX by selecting **New** from the **Project** menu. This creates a new project in a temporary working directory. No permanent copy of your project will be created until you save it. At that time, VX- REXX will prompt you for the directory and filenames to use. The directory you specify must already exist, so you should create it before trying to save your project .

## Opening an existing project

You can open an existing project from the Workplace Shell by opening its project folder and double clicking on the project file (VRP). You can also open projects from within VX-REXX by selecting **Open** from the **Project** menu, which will prompt you for the path of the project. If you were editing another project and have not saved the project since the last change, VX-REXX will allow you to save the current project before proceeding.

## Saving a project

You can save an open project to disk at any time by selecting **Save** or **Save as** from the **Project** menu. Saving a project saves the source files and the project file. **Save** saves the files using their current filenames.

However, if you created the project using the **New** menu item and have not yet saved it, you will be asked to specify the path for the project as if you had chosen **Save as**. The **Save as** item displays a dialog allowing you to set the location and names of each of the project files. You usually keep all of the files for a project in one directory, but each file can be saved in a different directory.

## Running a project

You can test your project at any time in the design process by selecting **Run project** from the **Run** menu. The project is then compiled into an executable form in a temporary working directory (the original project is not changed). VX-REXX hides itself while the program runs.

It may be necessary to use the **Run** menu item in the **Options** menu to set some initial parameters for the test run, such as the working directory or initial arguments to pass to the program.

If a REXX error occurs while your program runs, it will stop running. VX- REXX will appear and describe the type and location of the error and let you edit the procedure that is in error.

You can also debug your project by selecting the **Debug project** menu item from the **Run** menu. See the chapter *Debugging a project* for more information.

Before testing or debugging a program, it is always a good idea to save the project.

## Making an executable

Once you are satisfied with your project, you can generate a standalone program by selecting **Make EXE file** from the **Project** menu. You will be prompted for a path and the project will be compiled into an executable file which you can then run on any machine on which the VX-REXX run time libraries have been installed . See the online *Read Me First* that came with VX-REXX for information about the run time libraries.

## Command line options

The VX-REXX design environment accepts the following options:

/o projectname Opens the project projectname.

/x exename Makes the executable file *exename* from the currently open project .

/m macroname Makes the macro file *macroname* from the currently open project .

/q Quit. If omitted, the design environment remains open.

These options can be used to automatically re-generate project executables from the OS/2 command line. For example:

```
vrxedit /o myproj.vrp /x myproj.exe /q
```

This example starts VRXEDIT.EXE (the design environment), opens the project MYPROJ.VRP, makes the executable file MYPROJ.EXE from the project and then quits.

# Adding objects to a window

When you start VX-REXX, an empty project window appears to which you can add objects such as data entry fields, push buttons and list boxes. This chapter shows you how to create and modify these objects to design your application user interface.

## Introduction to objects

An object is a basic building block for your application interface. An object consists of *properties* (its data), and *methods* (the functions it can perform) . Your program interacts with objects by creating them, setting and getting their properties, and invoking their methods. Objects interact with your program by sending it *events* to which the program responds.

The PushButton is a typical object. You create one by drawing it in a window. You can set the text on it by setting its Caption property. You specify what happens when the button is clicked by defining a Click event routine. The event routine is run when the button is clicked.

Some special objects, for example menus and windows, are created with commands on a window's pop-up menu and on the **Windows** menu. These objects are described in separate chapters (*Adding menus to a program*, *Multiple file projects*, *Secondary windows*).

Other objects are invisible. You cannot create or see them but you can use them; these objects are created for you. One of them is the Application object. You can use it to pass information from one file to another within an application, and to start and control execution threads. The Screen object is another such object. These objects are described in the online *VX¨REXX Reference*..

## Creating objects

When VX-REXX is started on a new project, an empty project window appears . In Figure 24 the window containing rows of dots is the project window. You create the visual interface to your application in this window.

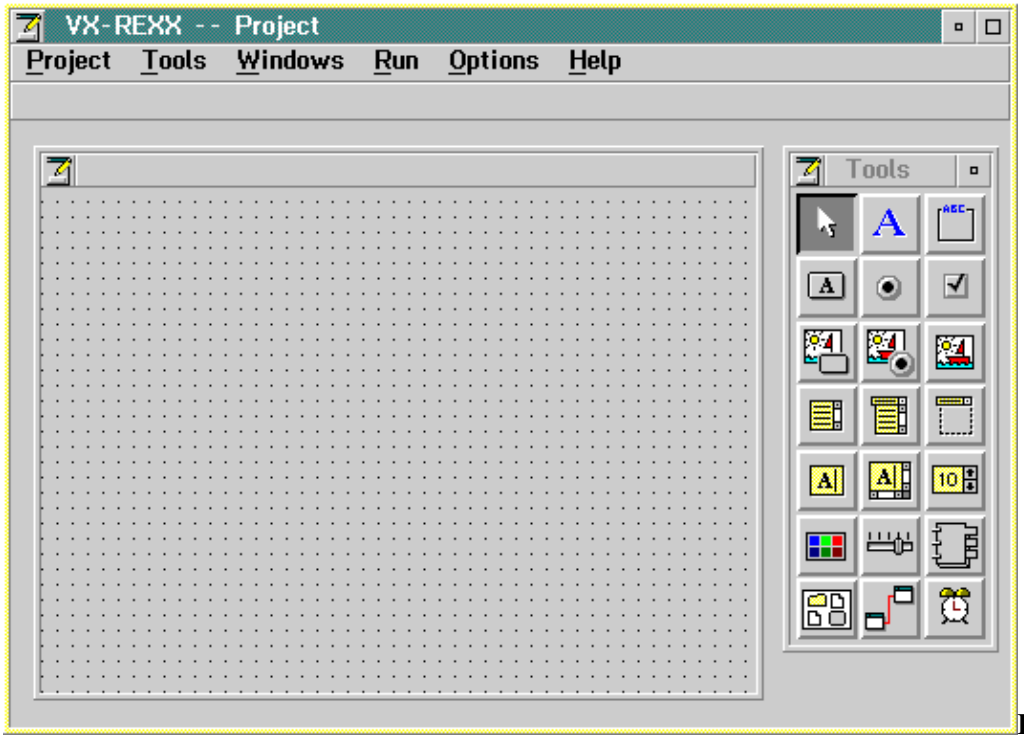


**Figure 24** The VX-REXX design environment

You begin to create an application by adding objects to the empty project window. The **Tools** menu contains the objects you can use to design your application interface.

To add one of these objects such as a push button to the project window, proceed as follows:

1.Click on the **Tools** menu and choose the object from the list.

2.Position the mouse pointer in the project window where the upper left corner of the object is to be placed.

3.Hold down mouse button 1 and drag the mouse until the outline of the object is the size you want.

4.Release the mouse button.

Use this procedure to add a push button to the upper left hand corner of the empty window. Your project window will look similar to Figure 25.

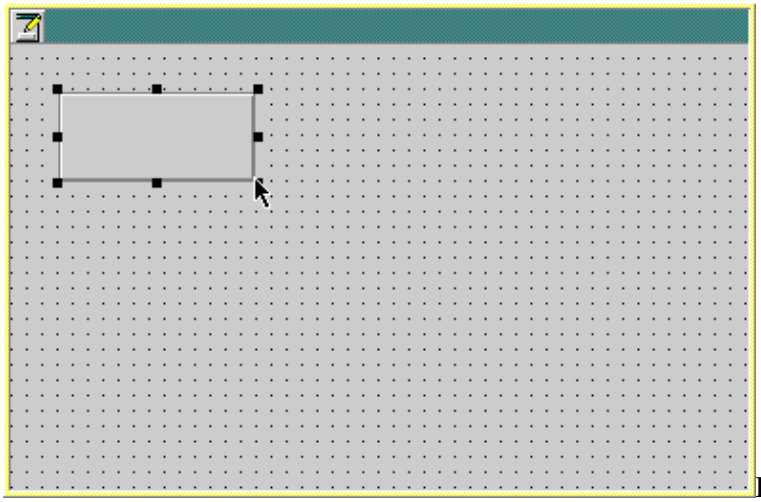

**Figure 25** Object added to a project window

## Using the tool palette

Each of the objects listed in the **Tools** menu has a corresponding icon in the tool palette as in Figure 26. This allows you to select objects more quickly .

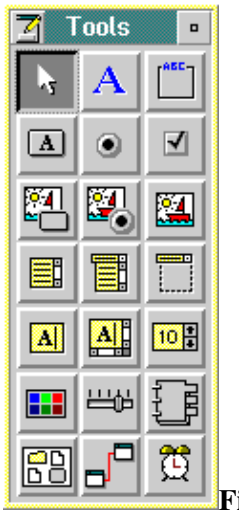


**Figure 26** Tool palette

The tool palette is displayed on the right side of your project window when you start VX-REXX. To add an object using the tool palette, click on the object icon to select it and then add the object to your window by positioning and dragging the mouse.

Unless you close it, the tool palette will remain displayed during the current session. When you become familiar with the icons, you may find that the tool palette is more convenient and faster to use than the **Tools** menu.

# Sizing and moving an object

After an object is created, you may find that it is not the exact size that you want or that it is not positioned correctly. You can easily resize and move objects using the mouse.

To resize an object, select it by clicking on it with mouse button 1. When you select an object you will see small black squares located on the corners and sides as in Figure 27.

**Figure 27** Sizing handles

These black squares are called sizing handles and are used to resize the object. The handles at the corners of the object let you change the height and width simultaneously. The handles in the middle of the sides let you resize one dimension at a time.

For example, to enlarge both the width and height of an object:

1.Position the mouse pointer on the bottom right corner sizing handle . You are positioned correctly when the mouse pointer changes to an arrow with a point at both ends.

2.Use mouse button 1 to drag the mouse down and to the right. Notice the outline for the new size of the object.

3.Release the mouse button. The object will be redrawn to the new size.

You can use a similar technique to resize the project window. Windows do not have sizing handles as objects do, but if you move the mouse pointer to the edge or corner of the window you will notice that the pointer changes to an arrow with a point at both ends. You can then resize the window by dragging the mouse using mouse button 1, until the window is the size that you want.

To move an object, position the mouse pointer anywhere within the object and use mouse button 2 to drag it to the new location. If you want to move two or more objects at the same time you must select them both. To do this, position the mouse above and to the left of the first object and, using mouse button 1, drag the mouse to enclose all of them in a box. For example, to move two push buttons together as in Figure 28:

1.Position the mouse pointer slightly above and to the left of the first push button.

2.Use mouse button 1 to drag the mouse down and to the right, surrounding both of the objects.

3.Release the mouse button.

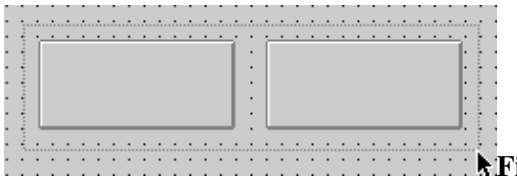

**Figure 28** Selecting two objects

Both of the objects within the outlined region will be selected and their sizing handles visible. The sizing handles will be gray rather than black. You can now move both push buttons as you would one object by positioning the mouse on either object and dragging the objects to the new location.

You can also select multiple objects by pressing and holding the **Ctrl** key while clicking on the objects that you want to select. Use this technique to select objects that cannot be enclosed in a boxed region.

## Selecting objects

There are two ways of selecting objects: *marquee selection* and *swipe selection*. You have already seen marquee selection in a previous section -- this technique involves dragging a rectangle around an object or a group of objects. When you release the mouse, all objects in the rectangle are selected.

You can also select objects by swiping the mouse across a number of objects while holding down the mouse button. This is called swipe selection. The following steps show you how to swipe select the two push buttons from the previous section.

1.Position the mouse over the first push button.

2.Press and hold down mouse button 1. The first push button will be selected.

3.Without releasing the mouse button, move the mouse over the second push button. Now both push buttons will be selected.

4.Release the mouse button.

## Copying and deleting objects

You can remove and copy objects in your project window with the **Cut**, **Copy**, **Paste**, and **Delete** commands found in the object pop-up menu. **Cut**, **Copy** and **Paste** use an internal clipboard to copy objects within a window or between windows within one project.

For example, to add three identical push buttons to your project window :

1.Create the first push button.

2.Click mouse button 2 anywhere within the object to display the object 's pop-up menu.

3.Choose **Copy**. A copy of the push button will be added to the clipboard.

4.Click mouse button 2 on the location in your program window where you want the copy to be placed. The window pop-up menu will appear.

5.Choose **Paste**. The top left corner of the pasted object will be located where you clicked in the program window.

6.To add the third push button, repeat steps 4 and 5.

Using cut, copy and paste, objects can be moved between project windows within an application. You will learn how to create applications with multiple windows in the chapters *Multiple file projects* and *Secondary windows*.

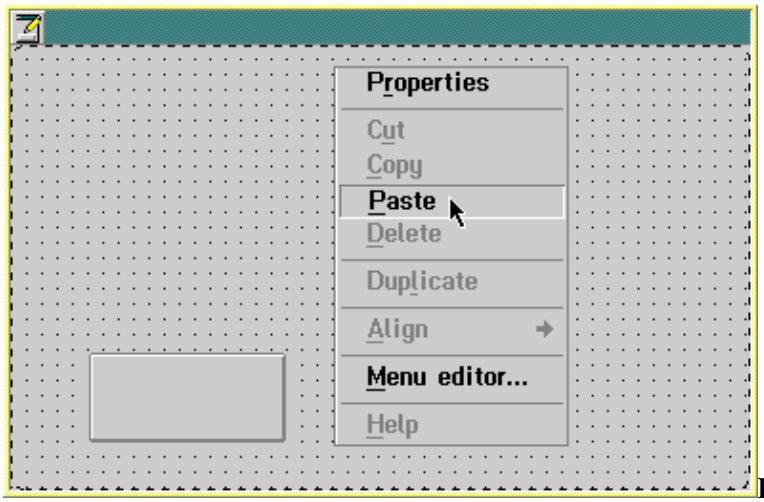


**Figure 29** Copying objects

**Delete** removes an object from a window. It cannot be pasted back. For example, to remove an object from your project window, move the mouse pointer over the object and click mouse button 2. Choose **Delete** from the pop-up menu as shown in Figure 30.

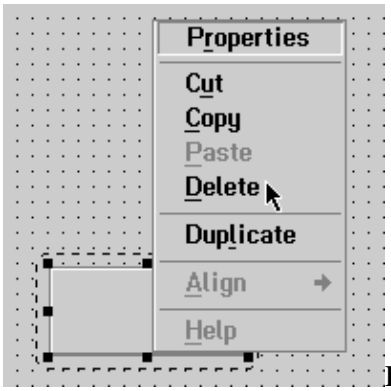


**Figure 30** Delete an object

The object will be removed from the project window.

# Duplicating an object

**Duplicate** creates a duplicate copy of an object. The objects are identical except for their location in the window. The duplicate object is created slightly offset from the original.

You should duplicate objects only when you want them to share the same event code. For example, you may want to use duplicate push buttons for the numeric buttons in a calculator program.

For example, to add three identical push buttons to your window:

1.Create the first push button.

2.Click mouse button 2 anywhere within the object to display the object 's pop-up menu.

3.Choose **Duplicate**. The duplicated object will appear offset from the original object.

4.Use mouse button 2 to drag the new object to its new location in your project window.

5.To add the third push button, repeat steps 2, 3 and 4.

When you duplicate an object, the clipboard is not used; the clipboard contents remain unchanged.

# Aligning objects

The background of the program window is covered with a pattern of regularly spaced dots. These dots form an alignment grid that you can use to ensure that objects are aligned vertically or horizontally. The grid is used whenever objects are created, resized or moved. For example, as you add an object to a window, the size of the outline changes in 'jump-steps' rather than in a smooth motion.

The grid helps you align objects when you are moving or resizing them. The **Align** command in the object pop-up menu lets you align objects more easily. You can use this command to align two or more objects on the left or right, at the top or bottom, or vertically or horizontally on the centre of one of the objects .

For example, if you have created a window with three push buttons you could align their left edges, as follows:

1.Select all three push buttons either by pressing the **Ctrl** key while clicking on each object or by enclosing the objects in a box using mouse button 1.

2.Click mouse button 2 anywhere within one of the buttons to display its pop-up menu. This button will be the target for the alignment.

3.Choose **Align** from the pop-up menu. The **Align** menu will be displayed as shown in Figure 31.

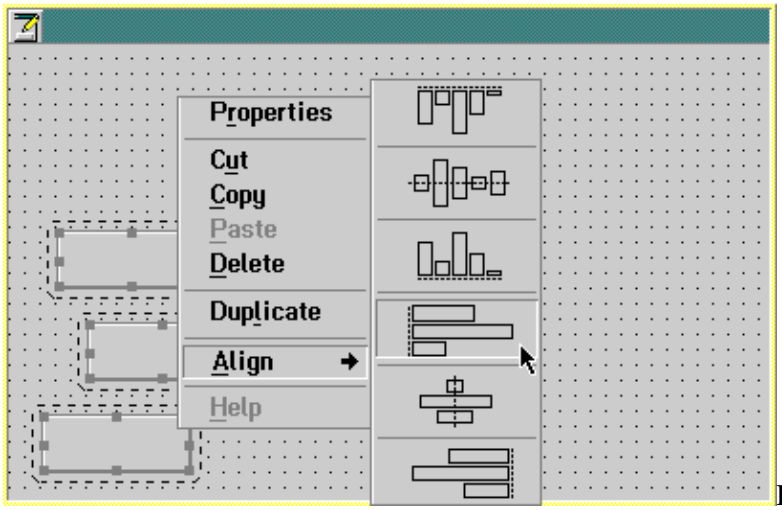


**Figure 31** Align menu

4.Click on the icon that shows a vertical dotted line to the left of three fields.

The objects will be aligned on the left side of the target push button, as in Figure 32.

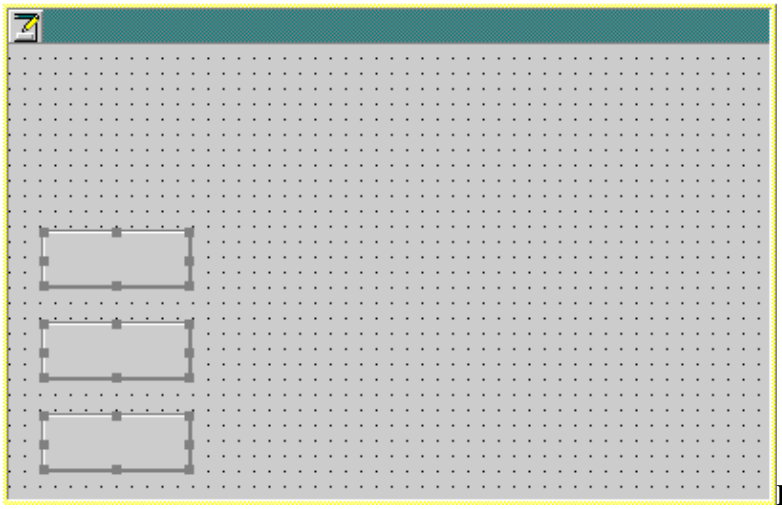

**Figure 32** Push buttons aligned

## Grouping objects

VX-REXX objects exist in groups. For example, all objects added to a window belong to a group. The GroupBox object allows you to create other groups within a window. Any object that can be placed in a window can be put in a group box, including other group boxes. If you draw a group box around an existing set of objects, the objects will be added to the new group box. Objects may also be added by cutting and pasting them into the group box.

Group boxes are most commonly used to group radio buttons. Since only one radio button can be set at one time, you must group them if you have more than one set of radio buttons in a window. This allows the groups of buttons to function independently.

The group boxes in Figure 33 contain two groups of independent radio buttons .


**Figure 33** Sample group boxes

You can also use group boxes just to emphasize the relationship among the grouped objects, or to enhance the appearance of your window. Also, since objects in a group move together when you move the group, you can use groups to simplify designing windows.

## Moving objects to the front and back

Objects on a window or in a group box are drawn in a particular order, sometimes referred to as the *z-order* or *sibling order*. The sibling order is only important if two objects in the same window or group box overlap. If both objects have the ClipSiblings property set to 1, they draw themselves so that the object with the lower sibling order is on 'top' of the other object.

The sibling order of an object is changed at design time using the **Bring to front** and **Send to back** commands on the object pop-up menu. (If an object is already at the top of the sibling order, the **Bring to front** command is removed from the menu; similarly, if an object is at the bottom of the sibling order, the **Send to back** command is removed from the menu.) You may also select several objects and send them to the front or back as a group.

To change the sibling order of an object at run time, use the SiblingOrder property.

Remember that changing the sibling order of an object may appear to have no effect unless the ClipSiblings property is set. ClipSiblings is not set by default , because some objects (notably the DropDownComboBox object) do not behave as expected if it is set.

# Changing object properties

Each object in your project's user interface has properties which determine both how it appears and how it functions. You can change property values at both design time and run time. This chapter shows you how to change property values when designing the user interface.

When an object is created, default values are assigned to its associated properties. At design time, you can change the values using the object's property notebook. For example, you can add a caption to a push button. You can also examine and change the values of many properties while the application is running. For example, you may want to disable a push button once it is pressed. VX-REXX provides functions to query and change property values at run time. Run time procedures are described in the chapter named *Programming with objects*.

## Property notebook

When you add an object to your project window, initial values are set for its properties. You change an object's properties using its property notebook, which is opened by selecting the **Properties** item from the **Open** menu in the object's pop-up menu. To open an object's pop-up menu, click mouse button 2 anywhere within the object. The property notebook for a push button is displayed in Figure 34.

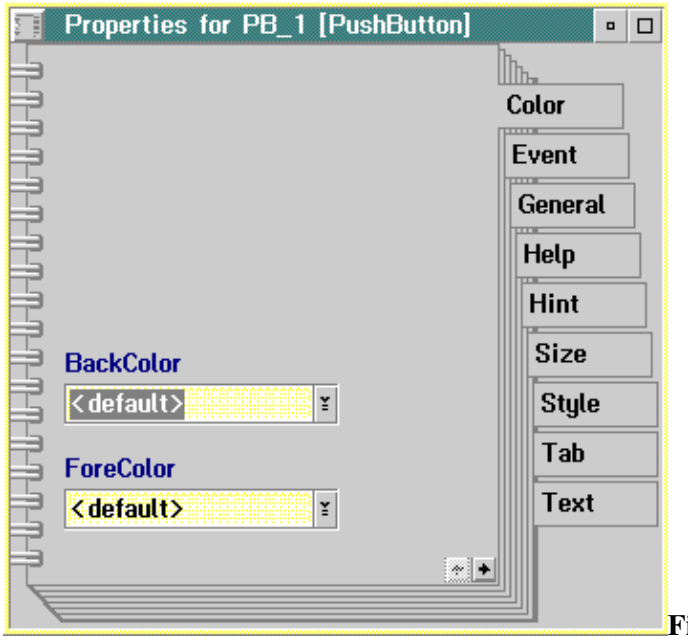


**Figure 34** A push button's property notebook

In this example the first page of the notebook shows the values of the color properties associated with the push button. The index tabs are the names of all of the property groups for this object.

A few properties, such as Name, are common to every object. Others, such as BackColor and ForeColor, apply only to certain objects. Only properties which apply to the current object are shown.

You can open the online information for any property by clicking on its value in the property notebook and then pressing **F1**.

## Changing property values

You can change a property value either by entering a value or by choosing an option from the property's notebook page. For example, to change the caption property of a push button, open its property notebook, click on the **Text** index tab, then click on the entry field for Caption and type the value, such as **Push Me**, as in Figure 35.

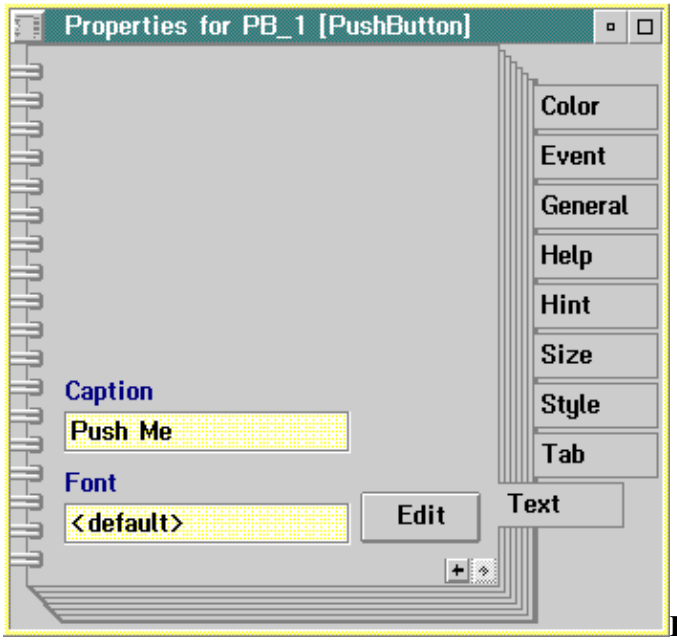


**Figure 35** Text properties

When you press **Enter** the caption appears on the button.

A property change is applied to the object as soon as you make the choice . You can leave the notebook open but your screen will be less cluttered if you close it when you have finished making property changes.

Like the objects on a window, the window itself has properties. For example, the title which appears on the title bar corresponds to the window's Caption property. There is no caption set for a window when you first create it .

To add or change a caption:

1.Click mouse button 2 on an empty location in the window. The window object's pop-up menu will be displayed.

2.Click on the **Open** menu and then the **Properties** item. The window 's property notebook will be displayed.

3.Choose the **Text** index tab.

4.Click on the Caption entry field and enter a value such as **First Application**.

5.Press **Enter** to make the change.

You will notice that the caption is displayed in the title bar of the window .

Some properties, such as the caption, are text strings. Some others can only have two values, 1 or 0. These appear as check boxes, as shown in the following example of the Border property for the push button.

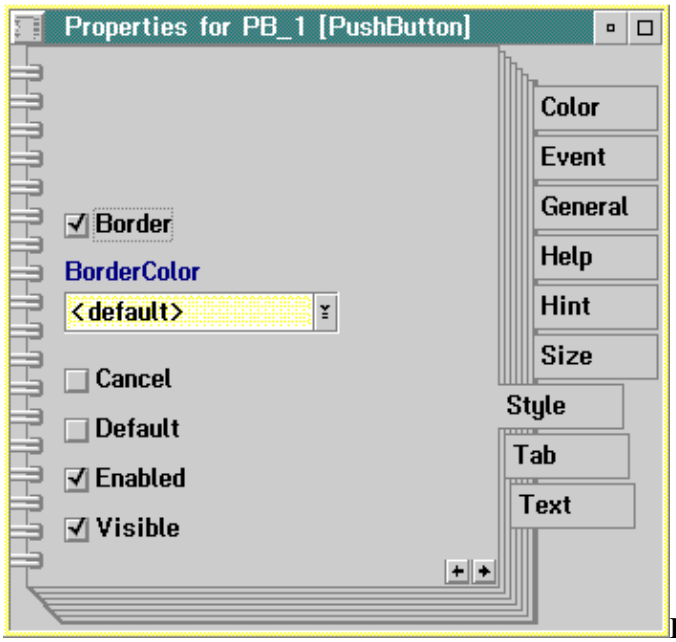


**Figure 36** Making a choice in a notebook

If you examine the push button's Border property on the **Style** page you will find that it is checked, indicating that the push button has a border. If you click on the check box, the border on the push button is removed. The push button caption is all that remains visible. Most of the time you will want the border to be set for a push button.

Other properties are set by selecting a value from a list. For example, Figure 37 shows the list used to change the BackColor property on the **Color** notebook page.

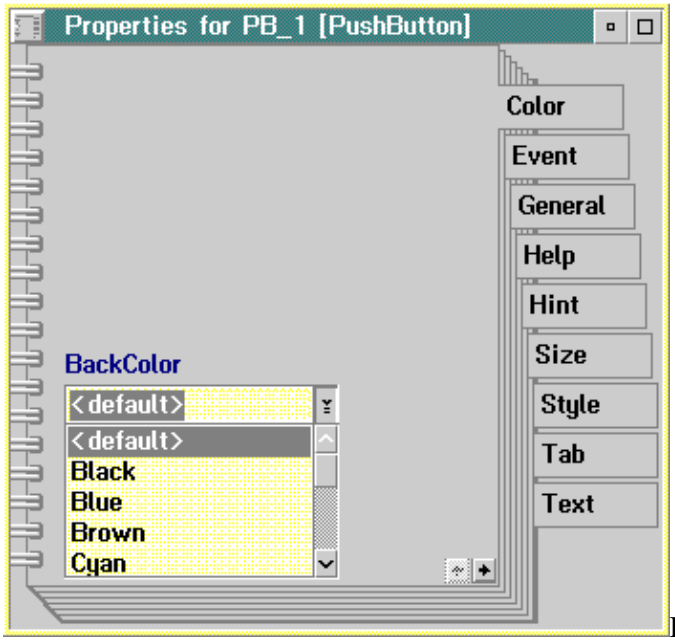


**Figure 37** Choosing a property value from a list

In some cases, a separate dialog is used to set a value. For example, the Font property is defined by a number of characteristics including a font name, style and size. You select a font by clicking on the **Edit** button on the **Text** page of the property notebook to display the **Font** window as in Figure 38.

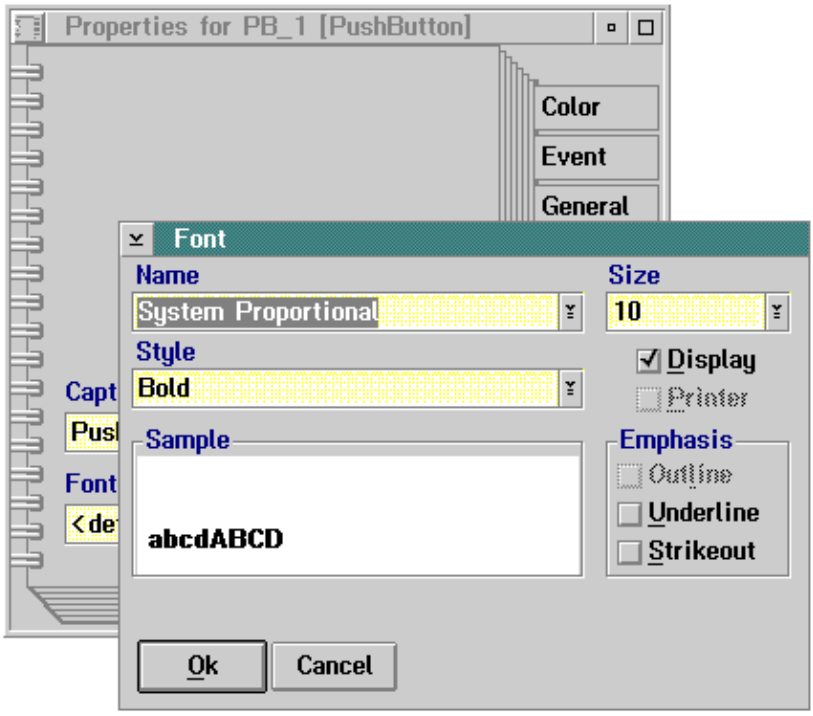


**Figure 38** Changing the Font property

You can select values from the drop down lists and the check boxes to choose the appropriate font. The changes are reflected in the **Sample** box. The font changes in the project window when you click on **Ok**. If you decide to not change the font, you can click on **Cancel**.

# Properties of copied and deleted objects

All property values for an object are cut, copied, pasted, duplicated or deleted when you cut, copy, paste, duplicate or delete an object in your window.

When you cut or copy an object it is saved to the clipboard along with its properties. You can then paste the object into the current window or into another window within the same project.

When an object is pasted, the following properties are changed automatically so that the values are unique for the new object:

oTop and Left are changed to the location of the pasted object.

oTabIndex is changed to add the new object to the end of the tab order.

oName is changed if necessary to give the new object a unique name.

The other properties remain the same.

When you duplicate an object, it is not copied to the clipboard. Any information previously copied to the clipboard remains. Duplicating an object is the same as copying and pasting except that the name of the new object remains identical to the original object. Because they have the same name, duplicate objects share event routines; i.e., the same event routine is used for both objects .

If you cut or copy one of a set of duplicate objects to the clipboard and then paste it, its name will change. It will no longer be a duplicate object . If you select several duplicate objects and then copy them to the clipboard, their names will change to the same value when pasted. They will be a new set of duplicate objects.

When you delete an object both the object and its properties are deleted. You cannot paste it back in the project window.

# Direct editing

Direct editing may be used to at design time to edit the text of an object with a Caption or Value property. To start direct editing, click on the object while pressing the **Alt** key. An entry field will appear. Enter or modify the text. Click outside the entry field to terminate the editing and save the new text, or press the **Esc** key to cancel the editing.

# Adding and modifying routines

The third and final step in creating a program in VX-REXX is attaching the REXX code. The previous two chapters described procedures for creating objects and setting their properties. This chapter describes the procedures to create and attach routines to the objects. It also describes the creation and use of general routines that are not attached to any particular object.

## Introduction to routines

There are two types of routines used in VX-REXX: event routines and general routines. Event routines are called automatically when the user triggers the event, such as clicking the mouse on an object, while general routines are called explicitly by other routines within your application.

### Event routines

Similar to properties, each object has a set of events associated with it . Each object type has a unique set of events to which you can attach REXX routines. These routines are called *event routines*. When an object is created, its event routines are empty. If an action should be executed when the event is triggered, you add code to that event routine.

At run time, when you trigger an event for an object, the corresponding event routine is run automatically.

### General routines

Programs usually have other collections of routines which are not directly tied to events. These routines are called *general routines*. General routines allow programs to be factored into logical units and allow you to reuse common code segments.

Event and general routines are both just normal REXX routines. Within one file (window file or code file), any routine can call any other routine. However , it is best not to call event routines directly as this will make your code harder to understand and maintain. Also, if you delete or rename the object corresponding to an event routine the event routine will be deleted or renamed. Instead of calling an event routine directly, you should factor out the common code and place it in a general routine and call it.

## Introduction to sections

VX-REXX adds the concept of sections of routines within a file to standard REXX. A standard REXX program consists of a file containing routines. You edit the entire file at once. For very large REXX programs, maintaining this file can be difficult. VX-REXX uses sections within a file to group routines into more manageable components. When using VX-REXX, you can choose to edit your application section-by-section or to edit the entire file.

A section is a named set of related routines. The name of the section is normally the same as the name of the first routine in the section, though this is strictly only necessary for sections containing event routines. There

can be multiple routines in a section. A section is not a callable entity in the REXX language, it only provides modularization of the REXX routines within a file. Sections have no effect on the scope of names; all names are global within the containing file.

A new section is created automatically whenever a new event routine is created. A section statement in a REXX file has the following form.

```
/*:VRX name */


The name would be the same as the label of the first routine in the
section .
```

# Editing sections

## Editing event routine sections

You can think of an event routine as another type of property that is associated with an object. Creating or changing an event routine is similar to setting other properties.

Use the following procedure to create a new section for an object's event routine or to edit an existing event routine.

1.Click mouse button 2 anywhere within the object to view the pop-up menu .

2.Choose **Open...** and then choose **Properties** to display the object's property notebook.

3.Choose **Event** to view the list of possible events for the object.

4.Select the event name from the list of items and click on **Open**, or double click on the name.

This opens the VX-REXX section editor which you can use to create or edit the event routine.

5.Enter the REXX instructions.

6.Close the section editor, if you are finished with it.

When you open an event routine which has not been previously created, VX-REXX automatically generates a new section which contains the event routine name and a REXX **return** statement similar to the following:

```
/*:VRX PB_OK_Click */
PB_OK_Click:

return
```

```
You need only fill in the body of the routine. If the event routine
already exists, the section which contains the existing routine is
loaded.
```

## Event routine names

VX-REXX generates names for event routines. An event routine name is the object's name followed by an underscore followed by the name of the event. For example, the click event routine for a push button with the name PB_OK would be PB_OK_Click.

## General routines

General routines are not automatically created and named by VX-REXX. You create them as needed using the **Sections** window. The next section describes how to use this window.

## Using the section list

The **Sections** window lists all of the sections in a file. From the **Sections** window, you can create new sections containing general routines, and edit and delete existing sections. This provides an alternate way to edit sections containing event routines. You must always create event routines using the property notebook.

To open the **Sections** window, select **Section list** from the **Windows** menu. The section list will remain open until you close it.

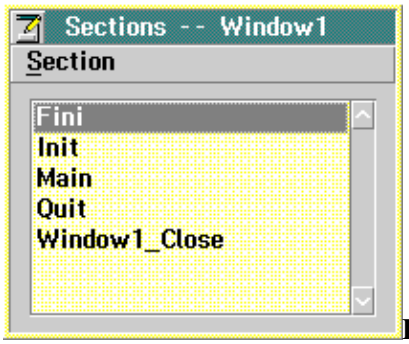


**Figure 39** The Section list

## Creating a new section

The following procedure creates a new section.

1.In the **Sections** window, choose **New** from the **Section** menu.

A dialog will prompt you for the name of the new section.

2.Enter the name of the new section, for example, DoIt.

3.Click the OK button.

The new section name, in this case DoIt, will be added to the section list . Also, a section editor will be opened containing the new section. VX-REXX automatically generates a skeleton routine consisting of the routine name (which is the same as the section name), and a REXX return statement. You need only to fill in the body of the routine. The new routine will look like this:

```
/*:VRX DoIt */
DoIt:

return
```

```
4.Enter the REXX instructions.

5.Close the section editor, if you are finished with it.
```

## Default section header

A default section header may be defined using the **Editor options** dialog. To open the dialog, select the **Editor...** item from the **Settings** menu under the **Options** menu. One of the groups on this dialog defines the default header to use when creating new sections. This header is a string which is appended to the end of the first label in the section. For example, if the default header is **procedure**, sections will be created in this form:

```
/*:VRX DoIt */
DoIt: procedure

return
```

```
Headers are only used if Create new sections with this header is checked.

If a default header is defined, it applies to any new section created in
any project.
```

## Editing a section

To edit an existing section:

1.In the **Sections** window, select the section name from the list of names .

2.Choose **Open** from the **Section** menu, or double click on the name.

This opens a section editor containing the section.

3.Enter or change the REXX instructions.

4.Close the section editor, if you are finished with it.

### Deleting a section

To delete a section:

1.In the **Sections** window, select the section name from the list.

2.Choose **Delete** from the **Section** menu.

The section name will be removed from the section list and the related routines will be deleted from the program. To delete an event routine, you delete its section.

## Using the section editor

Sections are entered and modified in an editor. You can choose to edit sections using an external editor, such as the OS/2 Enhanced Editor, as described in the next section, or using VX-REXX's built-in section editor.

The built-in section editor provides standard editing capabilities including **Cut**, **Copy**, **Paste**, and **Delete** commands. The **Cut**, **Copy**, and **Paste** commands use the OS/2 clipboard so you can move code between applications. The section editor also provides **Search**, **Replace** and **Go to line** commands.

### Searching

The **Search** command searches for a target string starting from the current position in the section to the end of the section. The following steps describe how to use the **Search** command.

1.Select **Search** from the **Edit** menu. The **Search** dialog will be displayed.

2.Enter the string you want to find.

3.Click on the **Find** button. The next occurrence of the target search string will be highlighted.

4.To find the next occurrence, repeat step 3.

5.Click **Cancel** to close the search dialog.

### Replacing

The **Replace** command searches for a target string starting from the current position in the section to the end of the section. You can replace the target string or continue to the next occurrence of the target string. The following steps describe how to use the **Replace** command.

1.Select **Replace** from the **Edit** menu. The dialog will appear as in Figure 40.

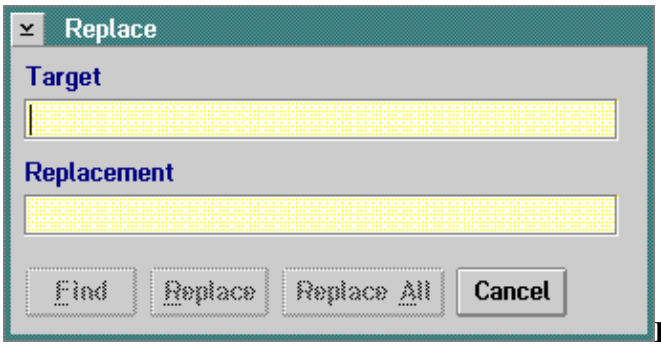


**Figure 40** Replace dialog

2.Enter the string you want to find.

3.Enter the replacement string.

4.Click the **Find** button. The editor will find the next occurrence of the target string.

5.To replace the current occurrence, click the **Replace** button. The string will be changed to the new value.

6.To find and replace the next occurrence, repeat steps 4 and 5.

7.Click **Cancel** to close the search dialog.

The **Replace** command can also replace all occurrences of the target string in the section with the replacement string. The following steps describe how to replace all occurrences.

1.Position the edit insertion point at the top of the file so that all occurrences will be found.

2.Select **Replace** from the **Edit** menu. The replace dialog will appear .

3.Enter the string you want to find.

4.Enter the replacement string.

5.Click the **Replace All** button. All occurrences of the string will be replaced with the new string.

6.Click **Cancel** to close the search dialog.

## Moving to a line

The **Go** command moves the edit insertion point to the beginning of a specified line number. The following steps describe how to use the **Go** command.

1.Select **Go** from the **Edit** menu. The **Go To Line** dialog will appear .

2.Enter the line number.

3.Click the **OK** button. The edit insertion point will be positioned at the beginning of the line that you entered.

### Accessing online information

You can open online information on VX-REXX and REXX topics directly from the editor in three ways.

The first way is to select a word with the mouse and then click mouse button 2 to bring up the editor pop-up menu. Select the **Help on** item to search the online information for the topic corresponding to the selected word.

The second way is to hold the **Ctrl** key while double clicking with mouse button 1 on a word. The word will be selected, and the topic will be searched for in the online help.

The third was is to open an online book by picking one of the books listed in the **Help** menu. The VX-REXX programmer's guide and reference, and the REXX language reference are all in this menu.

You can use these technique to view the reference information for all VX-REXX objects, properties, events, and methods as well as all REXX functions and keywords .

### Jumping to another routine

Similar methods can be used to quickly switch from one editor to another if the text in one editor refers to a procedure in another editor. Select the text and invoke the pop-up menu as before, but select the **View** item instead, or hold down the **Alt** key while double clicking on the word.

## Using an external editor

You can choose to edit any section using either the built-in section editor or an external editor such as the OS/2 Enhanced Editor. To use an external editor, use the **External editor** menu item or push button instead of the **Open** menu item or push button.

You can set the external editor to the editor of your choice. By default, the external editor is set to the OS/2 Enhanced Editor. Use the following procedure to use a different external editor.

1.Select **Settings** from the **Options** menu.

2.Select **Editor** from the **Settings** menu.

3.Enter the command to be used to start your editor.

4.Check the appropriate radio button.

5.Click the **OK** button.

The command you enter is interpreted by REXX, so enter the command as you would enter it in a REXX command file, including any required quotation marks. You can enter a single command or enter a number of statements separated by semicolons. You can also call an external REXX command file by using the **call** instruction in your command. You can even use the **say** and **trace** instructions if you call VRRedirectStdIO first.

There are three variables available that you can use in your command:

VREFileName The path and name of the file to edit.

VRELineNumber The line in the file where the insertion point should be placed .

VREWindow The internal name of the VX-REXX primary window (you can use this as the parent window if you invoke a dialog in your procedure).

The default command starts the OS/2 Enhanced Editor (EPM):

```
'start epm.exe' VREFileName '''VRELineNumber'''
```

The OS/2 **start** command is used to start the editor in a separate process so that VX-REXX and the editor both run at the same time.

**Note:** When using an external editor you must be careful to save your work before saving or running your project. This is because VX-REXX cannot command your editor to save its contents when they are needed. If your current work is not saved the project will be saved or run with the old code. The updating of sections is handled automatically when you use the built-in section editor.

## Event routines of copied or deleted objects

The cut, copy, paste, duplicate, and delete commands operate on an object as a whole, which includes the object, its properties, and sections containing its event routines. For instance, deleting an object removes the object, properties of the object, and sections of the object (all sections that start with the name of the object).

The paste command checks for a duplicate object name. If the object name is unique (e.g. the object was cut), the object, its properties, and its events are pasted as is into the program. If an object with the same name already exists (e.g. the object was copied), the Name property is automatically renamed, as described in the previous chapter, before the object is pasted into the program. The sections and event routines for the object are also renamed.

For instance, a push button named PB_1 would have a click event routine named PB_1_Click in a section named PB_1_Click. Copying this object using a copy and paste operation would generate a new push button with a new name, such as PB_2, with a click event routine named PB_2_Click in the section named PB_2_Click.

The duplicate command copies the object, its properties and its sections. However, unlike pasted objects, the name of the new object remains the same as the original object. Any event routines created for one of the duplicate objects are shared by all of the duplicate objects. If you delete an event routine from the section list, it is deleted from all of the duplicate objects.

If you rename one duplicate object, it will no longer share any of the event routines. You must create new ones for that object. However, if you copy a duplicate object to the clipboard and then paste it, the pasted

object will have a new name and the events associated with it will be copied and renamed. You can also select and copy a set of duplicate objects to the clipboard. When you paste them, all of them will have the same new name and share the copied events.

# Sharing sections

A shared section is a file containing one or more routines that are to be shared by several files in a project (see 'Multiple file projects') or by files in different projects.

## Adding shared sections

To add a shared section, select **Add...** from the **Section** menu in the **Sections** window. Enter the name of the file containing the routines that are to be shared. Absolute or relative file paths may be used. The shared section will be added to the front of the section list, as in:

```
<d:\vxrexx\projects\shared\common.vrx>
Fini
Halt
Init
Main
Quit
```

When the project is run, or an executable or macro command file is made, the contents of the shared section will be appended (added to the end) to the file.

## Editing shared sections

Shared sections are edited like normal sections: double-click on the section name or select the section and choose **Open** from the **Section** menu.

You may create new shared sections using your favorite text editor or using the section editor. To create new files using the section editor, simply create a new section in the current project, select all the text in the section editor, export it to an external file, delete the section and add the new file as a shared section.

## Deleting shared sections

Shared sections are deleted from the current file by choosing **Delete** from the **Section** menu. The shared section is not actually deleted, but simply removed from the current file.

## Search order

If a relative path is used when adding a shared section to a file, the following search order is used to locate the shared section:

1.The project folder.

2.The paths specified in the VXINCLUDE environment variable, if it exists .

3.The directory specified in the VXREXX environment variable.

# Programming with objects

To create a VX-REXX application you position objects on a window and attach event routines to the objects. The routines attached to the objects are performed when the application is running and the user interacts with the objects. Previous chapters showed you how to add objects and code to a window. This chapter gives an overview of the structure and execution of VX-REXX projects and explains the general techniques used in writing event routines.

## The structure of a VX-REXX project

### Code files

Not all VX-REXX applications include a graphical user interface. A simple VX-REXX project can consist of a single code file that contains a standard REXX program. A code-only application can be created by selecting **New code file** from the **Project** menu after creating a new project. (A new project is created when **New** is selected from the **Project** menu or when VX-REXX is started by double clicking on its icon ).

A new code file consists of an empty Main routine in a Main section. The code in the Main section is placed at the start of the REXX file and so is the code that is run first when the file is invoked. Since a section can contain many routines, all of the code in a project can be placed in the Main section. However it is easier to edit and maintain a large project if you use several sections.

The advantage of multiple sections is that they allow you to edit your program in smaller pieces. However, multiple sections have no effect on the way your program is run: all of the code runs as a single REXX file. This means that routine and variable names are subject to the usual rules for a single REXX file. Since routine names are global, any routine can call any other routine and the names of all routines must be unique. Similarly, because variable names are global, any variable can be used or set in any routine, unless the REXX keyword **procedure** is added to the routine label.

### Window files

Normally VX-REXX applications do include a graphical user interface. For example, a new VX-REXX project consists of a single window file. There are two parts to a window file: the window definition and the REXX code. The window definition lists the objects that form the user interface. The code is divided into sections for editing but is run as a single REXX file.

Every object has a name. A default name is generated when the object is created, but you can change the name by changing the Name property using the object 's property notebook. Object names are global within the window file: any routine in the file can refer to any object in the file using its name. Object names are used to get and set the value of an object's properties at run time. The name is also used to associate the object with its event routines. For example, the routine for the Click event of a push button with name PB_1 would be named PB_1_Click.

The code in a window file is divided into event routines and general routines .

The event routines are defined by entering code in an edit window opened on an event property of an object. The code in an event routine is standard REXX code, except that special VX-REXX functions are used to access the objects. These special routines are explained in *Interacting with objects from a program*.

In addition to event routines, a window file includes general routines that are called by the event routines or on start up. Putting code in general routines allows you to share code between objects and to keep general purpose routines separate from any object-specific code. For example, to handle two different events in the same way, create a general routine that is called from both event routines . This is better than calling one event routine from another as it reduces the chance that renaming or deleting an object introduces an error into the program.

## Predefined routines

A new window file consists of an empty window and a set of predefined routines: Main, Halt, Init, Window1_Close, Quit, and Fini (each routine is in a section of the same name). The routine called Main is the first routine that is run when the window file is run. It loads the window and processes its events. You should not modify the Main routine as its definition may be changed in future versions of VX-REXX. Instead, modify the other predefined routines. They have been provided to allow you to connect to the start up and shutdown of a window.

When a window file is executed, the Main routine gathers up any arguments passed to the window file and places them in the global compound variable **InitArgs** . **InitArgs.0** is set to the number of arguments, and each of **InitArgs.1** to **InitArgs.***n* is set to each of the *n* arguments in turn. If the window file is the main or only file in a project, the arguments passed to it are the arguments entered on the command line that started the project. (The command line itself is also available from the CommandLine property of the Application object). If the window file is called from another file, the first argument is assumed to be a window name . In this case, the window name is used as the parent window and the remaining arguments are placed in **InitArgs**. This is explained more fully in *Multiple file projects*.

The Main routine then loads the associated window and calls the Init routine . On entry to the Init routine, the window has been loaded but is not visible . The default Init routine sets the window to be visible and returns. You can add code to the Init routine to initialize your application. If your initialization is short you might like to complete it and then make the window visible. If you have a long initialization you might like to make the window visible right away , but disable it until your initialization is complete. See the online *VX¨REXX Reference* for information on the Visible and Enabled window properties.

After returning from the Init routine, the Main routine enters a loop which waits for an event and then runs the corresponding event routine. The events are processed one at a time. Events that occur while an event routine is running will be queued and processed when the current event routine is finished. If an event routine takes a long time to complete, the user can queue many events. If you want to prevent the user from creating events, you can disable the window until the current processing is complete. You may also want to start a separate thread to handle long computations to allow your event routine to return right away. See the _StartThread method in the online *VX¨REXX Reference* .for information on starting a separate thread.

The predefined routine Window1_Close is the event routine for the close event of the window. The event is triggered when the user selects Close from the window's system menu. This routine calls the Quit routine to shut down the window.

The event processing loop is stopped when the window's ShutDown property becomes 1. The Quit routine sets the window ShutDown property to 1. You can call the Quit routine from any of your routines to close the

window. You can also change the Quit routine so that it allows the user to confirm or cancel the close. If the user decides not to close the window, just return from the Quit routine without setting the ShutDown property, and your project will continue to run .

Note that if you determine in the Init routine that the project should quit right away you can call the Quit routine before returning. This will prevent the event loop from running at all.

After exiting from the event processing loop, the Fini routine is called. At this point it is too late to prevent the window from being closed, but the window still exists. This allows you to get the final value of radio buttons, entry fields, etc. so you can set a return value for the window file. The value returned by the Fini routine is the value returned by the window file. For this reason, the Fini routine must return a value. The default return value is 0 . See *Multiple file projects* for more information on the return value of a window file.

### Multiple files and windows

More complex VX-REXX projects consist of several files. These files can be any combination of code and window files. One file invokes another by calling it with the REXX **call** instruction or by invoking it as a REXX function. Each file in a project executes as a separate REXX program file. This means that all of the labels and variables in one file are hidden when executing another file . In addition all object names in one file are hidden when executing another. This allows you to use the same names in different files without fear of conflict . Data is passed between files using REXX arguments and return values, or by using the PutVar and GetVar methods of the Application object. Using multiple files is explained in *Multiple file projects*.

Each window file has its own event queue and event processing loop. This means that, if one file calls another, it is not possible to process the events from two windows in two different files at once. This is because only one of the event processing loops will be running. This is not a problem if the called window is a modal dialog that the user must finish before returning to the calling window. Sometimes, however, it is necessary to provide access to multiple windows at the same time. Although this can be dealt with by using multiple threads, it is usually more straightforward to handle this by defining multiple modeless windows within a single window file. One event loop dispatches events for all the windows so none are disabled. Also, since all of the event routines have access to all of the object names and variables, it is quite simple to share information between windows. See *Secondary windows* for more information.

## Interacting with objects from a program

A program interacts with an object by its events, properties, and methods . Events are messages the object can send to the program. The program can receive these messages and respond to them by executing an appropriate routine. The properties of an object are its data and state information. A program can get and set an object's properties, so it can read and change the object's state . The methods of an object are its functions. A program invokes a method of an object to cause the object to perform a function.

The following sections explain how to refer to an object, and how VX-REXX programs interact with objects by responding to events, getting and setting properties, and invoking methods.

### Referring to an object

Every object has a Name property which is assigned a unique value when it is created. The assigned name is the name or initials of the object type followed by a number generated by VX-REXX. For example, the assigned name of a push button may be PB_1. You can change the default name using the General page of the property notebook.

Object names are global within a file. Any routine in any section can refer to an object in the same file by its name. Object names are hidden between files. Objects in different files can have the same name without conflict. While it is not possible to refer to an object in another file by its name, it is possible to refer to objects in other files by using internal names, as explained below .

Duplicate objects are objects in one file with the same name that share the same event routines. They can only be created by using the Duplicate item on an object's pop-up menu. It is not possible to refer to an individual duplicate object by its name, as there is more than one object with that name. It is however possible to refer to duplicate objects by using internal names.

In addition to the Name property, every object has a unique internal name assigned when it is created. An object's internal name is always unique, and is unique over the entire VX-REXX project. Whenever an object name is required as a parameter to a VX-REXX function you can use either the value of the Name property (the external name), if it's unique, or the object's internal name, which is always unique.

The Self property of an object contains its internal name. It is not available at design time because the internal name of an object is only assigned when the object is created at run time. The Self property can be used to obtain the internal name of an object to pass to another file.

The VRInfo function can be used to return the internal name of the object that triggered an event. The VRInfo function can only be used within event routines as it returns information about the current event. It can be used within event routines to get the internal name of duplicate objects. A typical function call is:

```
object = VRInfo( 'Object' )
```

The VRWindow function returns the internal name of the current primary window (the window automatically loaded when the window file was invoked). It is most often used with the dialog functions which require the name of the window to be disabled during the dialog. The following shows VRWindow used with the VRMessage function.

```
call VRMessage VRWindow(), 'Time to go home!', 'Time reminder', 'I'
```

*Using dialog and system functions* documents the VRMessage function, among others.

## Renaming an object

If you are going to refer to an object in a program, you should give that object a more meaningful name -- for example **PB_Start** rather than **PB_1**. If you change the names, it will be easier for you to remember them and easier to read and understand the code.

When an object is renamed, references to that object are updated in the REXX code. The object must have a name of the form *prefix_suffix* where *prefix* is an alphanumeric string. The following strings are changed in the REXX code:

oLiteral strings
oLabels
o**call** instructions

For example, if the object **PB_1** is renamed to **PB_Start**, the following code

```
PB_1_Click:
if( VRSet( 'PB_1', 'Caption' \= '' ) ) then signal PB_1_Click_End
call VRSet 'PB_1', 'Caption', 'Clicked'
PB_1_Click_End:
return
```

```
is updated to

PB_Start_Click:
if( VRSet( 'PB_Start', 'Caption' \= '' ) ) then signal PB_Start_Click_End
call VRSet 'PB_Start', 'Caption', 'Clicked'
PB_Start_Click_End:
return
```

## Implicit and relative naming

Objects can also be referred to using *relative naming*, where the name that identifies the object is a combination of names belonging to the object and one or more of its parent objects. The names are separated using periods.

For example, consider a window **Window1** with a group box **GB_1** and a push button **PB_1** inside that group box. The push button can be referred to as:

```
PB_1
GB_1.PB_1
Window1.PB_1
Window1.GB_1.PB_1
```

```
If a second group box GB_2 containing another push button PB_1 is added
to the window, then PB_1 by itself is ambiguous, but the following are
not:

GB_1.PB_1
GB_2.PB_1
Window1.GB_1.PB_1
```

```
Window1.GB_2.PB_1
```

Two objects with the same name may exist within a single file, which can cause ambiguity. This is a common situation when the same secondary window is loaded twice or more by a project.

Each component of a relative name can be a simple name or an internal name .

VX-REXX also implements a form of relative naming called *implicit naming* which is useful for files that load two or more copies of the same secondary window . Implicit naming is on by default, but it can be turned off using the VROptions function. The internal name of the current event window (the window returned by passing the **Window** parameter to VRInfo) is implicitly prefixed to the name of an object to form a relative name. In other words, when implicit naming is on the following code:

```
call VRSet 'PB_1', 'Enabled', 0
```

is equivalent to the following code when implicit naming is off:

```
win = VRInfo( 'Window' )
call VRSet win || '.' || 'PB_1', 'Enabled', 0
```

If implicit naming is on and the object is not found, the object is searched for again without the implicit prefix.

## Responding to events

Events are usually caused by a user interacting with an object. For example a user can click on a button. Many objects handle events automatically. For instance, clicking a check box automatically toggles the state of that object for you. This includes checking or unchecking the box on the screen as well as updating the Set property. No programming is needed for this to happen. However, after updating the internal state, the check box will queue a Click event so that your program can respond to it.

There is usually more than one way to generate a particular event. For example, the Click event for a push button can be generated by clicking on the push button or by pressing the space bar when the push button has the focus. The object does the work of the low-level interaction with the user and only passes on ' interesting' events to the program.

The correct response to an event depends on the object, the event, and on the purpose of the program. The next chapter *Using objects* shows how to deal with some specific objects and events. In addition to standard REXX programming, the event routines set and get object properties and invoke object methods, as described in the next section.

## Getting and setting properties

Two functions are provided to get and set the properties of an object: VRGet and VRSet. Each property is identified by a name. Most properties appear in the object's property notebook, however some properties (such as Self) are not available at design time. It is always possible to get a property' s value, but some properties are read-only at run time and cannot be set.

The valid values for each property depend on the property. Typically a value must be a logical value (0 or 1), a number (such as for the size or position of an object), one of a set of values (e.g. **Red** for BackColor), or any text string (Caption). Property names and values that are one of a set of values are always case insensitive.

Property values are always stored and returned in canonical form. This means, for example, that logical values are always returned as 0 or 1, and values that are one of a set are always returned in mixed case, such as **Red**.

*Summary of properties* contains a complete list of property names. The online *VX¨REXX Reference* .contains a detailed description of each property.

The function VRGet returns the current value of a property. This function requires two parameters: the object name and the property name.

```
objectCaption = VRGet( 'PB_PushMe', 'Caption')
```

In this example, the value of the Caption property for the object named PB_ PushMe is assigned to the REXX variable **objectCaption.**

The VRSet function changes the current value of a property to the specified value. This function requires three parameters: the object name, the property name, and the new property value. More than one property can be changed at once by including multiple property names and their values as pairs of arguments . Changing more than one value at once can improve the performance of the change . For example, it is better to change the Height and Width properties together so the object will only be redrawn once. An example of VRSet follows.

```
call VRSet 'PB_PushMe', 'Caption', 'Push Me', 'BackColor', 'Red'
```

In this example, the value of the caption property for the object named PB_ PushMe is changed to **Push Me**, and the BackColor to **Red.**

You can check the results of VRSet as follows:

```
rc = VRSet( 'PB_PushMe', 'Caption', 'Push Me', 'BackColor', 'Red' )
```

The value returned in the variable **rc** indicates whether the property was set successfully (1 is returned) or not (0 is returned).

Some of the most common uses of run time property editing are:

oTo enable and disable push buttons. The following disables the push button named PB_PushMe.

```
call VRSet 'PB_PushMe', 'Enabled', 0
```

oTo set the value of an entry field. The following sets the value of the entry field named EF_Text to the value **<none>.**

```
call VRSet 'EF_Text', 'Value', '<none>'
```

oTo toggle a radio button or check box. The following changes the Set property of the radio button named RB_1 to show that it is selected.

```
call VRSet 'RB_1', 'Set', 1
```

## Using object methods

Each object type has special built-in operations which act upon the object . These operations are called methods. For a complete list of methods, refer to the chapter *Summary of methods*.

A method is invoked using the VX-REXX function VRMethod. This function requires at least two parameters, the object reference, the method name, and then any other parameters required by the particular method. In the individual method descriptions in the online *VX¨REXX Reference*., the other parameters are described as the method's arguments.

In the following example, the method named GetSelectedStringList is invoked for the list box object named LB_Color and passed the argument **color.**

```
call VRMethod 'LB_Color', 'GetSelectedStringList', 'color.'
```

The method GetSelectedStringList returns the selected strings from the list box LB_Color in the compound variable **color.**

# Using objects

This chapter explains how to use VX-REXX objects in your project. For the more common objects, it describes the object and gives programming instructions for certain object-specific tasks. The first part of the chapter shows you some operations that you can perform on most objects. They are: disabling and hiding objects, setting the input focus, and setting the tab order.

For complete information on all the VX-REXX objects refer to the online **VX- REXX Reference**.

## Common operations on objects

### Disabling objects

There are times when you want an object to be visible, but you do not want to allow the user do anything with that object. For example, if you have a window with an entry field and a push button you may not want the user to push the button before entering a value in the entry field. You want to *disable* the push button while the entry field is empty.

You can disable an object by setting its Enabled property to 0. You can enable an object by setting its Enabled property back to 1. By default, all objects are created with their Enabled property set to 1.

You should never disable the main window for your project! If you do, you will not be able to close the window or quit the project except perhaps by using the OS/2 Window List.

### Hiding objects

There may be times when you want to hide an object. You can control the visibility of any object using its Visible property. To hide an object, set its Visible property to 0. To show it again, set the property to 1. By default, all objects are visible.

### Getting and setting the focus

The object which is currently the target of any input is said to have the *focus*. When an object gains or loses the focus, an event occurs for that object . These events are GotFocus and LostFocus, respectively. Usually these events occur in pairs; the object that loses the focus receives a LostFocus event, then the object which is receiving the focus gets a GotFocus event.

You should **not** use the GotFocus and LostFocus events to perform user input validation. The reason is that these events can be caused by actions that have nothing to do with user input. For example, if the user activates a different application, then a LostFocus event is generated, regardless of whether the user was finished entering data or not. Use the Verify event instead, as described below.

You can use the SetFocus method to move the focus to a given object. Suppose you have written a dialog that validates the input received from a user. If the data is invalid, the program should alert the user, then set the

focus to the object which contains the errant data.

To determine which object (if any) in your project currently has the input focus, use the following code:

```
GetFocusObject: procedure
focus = VRMethod( 'Screen', 'GetFocusWindow' )
if( focus \= '' )then do
focus = VRGet( focus, 'Object' )
end
return focus
```

The routine returns the internal name of the object with focus, or a null string if no object in the project has the input focus.

### Input validation

A Verify event exists for the EntryField, DropDownComboBox, ComboBox, MultiLineEntryField and SpinButton objects. This event occurs when the input focus moves to another object *on the same window*, allowing you to validate the contents of the Value property and if necessary reset the focus. See the description for Verify for more details.

### Setting the tab order

In a window, the user can press the **Tab** or **Backtab** keys to move the focus from object to object. The order in which each object gets the focus is called the *tab order*.

You can set the tab order by setting the TabIndex property for each object in a given window. Initially, the object that has TabIndex set to 1 has the focus. When the user presses **Tab**, the focus moves to the object with TabIndex equal to 2, and so on.

If you do not want to include an object in the tab order, set its TabStop property to 0. Most objects have TabStop set to 1 by default. Radio buttons and group boxes are created with TabStop cleared. A radio button is only included in the tab order if it is selected so only one of the buttons in a group is included in the tab order. Group boxes are never included in the tab order.

The TabStop and TabIndex properties are set automatically when you add an object to a group or window.

To aid in setting the tab order of a window, a tab editor is available in the design environment. Click on the window with mouse button 2 and select the **Tab Editor** from the **Open** item on the pop-up menu.

## Windows

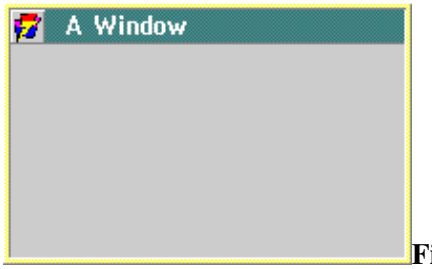


**Figure 41** A window

Users communicate with your project through windows. Each window presents a view of the project. More than one window can be used at the same time to show different views.

A window is a rectangular area on the screen. It consists of a frame and any number of other objects which appear in the window. The frame may be a simple border, or it may also include a title bar, a system menu, minimize, maximize and hide buttons.

## Setting the border type

A window can have one of three borders: dialog, sizeable, or thin. Use a dialog border for modal or system modal windows. Use a sizeable border if you want to let the user resize the window. Use a thin border for all other cases.

To set the border type, set the BorderType property. You must set this property at design time.

## Setting the caption

You can change a window's caption by setting its Caption property. The caption appears in the title bar. If the caption is longer than the width of the window, only the leftmost portion will be visible.

## Adding minimize, hide, and maximize buttons

By default a window is created without maximize, minimize, or hide buttons . To add a minimize button, set the MinimizeButton property of the window. To add a maximize button, set the MaximizeButton property. To add a hide button, set HideButton to 1.

A window can have either a hide button or a minimize button, but not both . As a result, if MinimizeButton is 1, you cannot set HideButton to 1 until you set MinimizeButton to 0.

If you include a maximize, minimize, or hide button, the corresponding items in the system menu are enabled.

You must set the MinimizeButton, MaximizeButton, and HideButton properties at design time. Setting them at run time will have no effect.

## Removing the system menu

By default, a window is created with a system menu. Since the system menu can be used to close a window, it is a good idea to always include one. If you remove it, you must provide some other object, such as a push button, to close the window. You need to call the Quit routine from the push button's Click event .

To remove the system menu, reset the window's SystemMenu property. You must set or clear this property at design time.

## Removing the title bar

By default, a window is created with a title bar. To remove the title bar , reset the window's TitleBar property.

## Resizing objects in windows

If you want the contents of a window to automatically resize themselves when the window is sized, you may be able to use the LayoutStyle property of the window . If the window contains a single object and some push buttons, set the LayoutStyle property to **SimpleDialog**. The window will then automatically resize the object it contains, and will place the push buttons appropriately. If the window has more objects on it, then the program must reposition or resize objects. An ideal place to do the sizing is the Resize event.

## Getting and setting the window state

You can determine whether a window is minimized or maximized by calling the VRGet function on the WindowState property. The value of the property will be one of **Minimized**, **Maximized**, or **Normal** (in the case that the window is neither minimized or maximized). Likewise, you can minimize, maximize, or return a window to normal size by using the VRSet function in conjunction with this property.

A Resize event is generated whenever a window is minimized, maximized or restored.

## Setting the OS/2 window list title

By default, only the main window of an application is added to the OS/2 window list. The WindowListTitle property is used to add a window to the window list or to remove it from the window list.

## Hints

Hints are single-line messages displayed in a special status area at the top or bottom of a window. When an object receives the keyboard focus or the pointer moves over the object, the hint for that object is displayed in the window 's status area. The hint text for an object is defined by the HintText property.

To add hint support, set the ShowHints property to 1 and set the StatusArea property to **Top** or **Bottom**. If hints are to be displayed for menu items, **Bottom** is recommended.

The font used by the status area is changed using the StatusFont property .

### Displaying a background picture

A background picture may be displayed in a window by setting the PicturePath property. See the 'Bitmaps, icons and resources' chapter for more information.

## Push buttons


A push button is used to initiate an action which is associated with that object.

There are two ways to press a push button:

oClick on the button using mouse button 1.

oIf the push button has the input focus, press the **SpaceBar** or the **Enter** key .

When it is clicked at run time a push button object gives the illusion of being pressed.

### Setting the caption

To change the caption for a push button, set the Caption property. If the AutoSize property is 1, the push button will automatically resize itself so that the entire caption is visible. If AutoSize is 0, and the caption is longer than the width of the button, the caption will be clipped.

### Creating a default button

In a window, the default button is automatically clicked when the focus is on another object (other than another push button) and you press **Enter**. To specify a default button, set its Default property to 1 at design time.

### Creating a cancel button

In a window, the cancel button is automatically clicked when you press **Esc** . To specify a cancel button, set its Cancel property to 1 at design time.

### Using the Click event

Typically, you will add code to a push button's Click event in order to perform an action. The following example shows the click routine for a button named PB_PushMe which toggles the background color of the

button between red and green .

```
PB_PushMe_Click:
color = VRGet( 'PB_PushMe', 'BackColor' )
if color = 'Red' then do
call VRSet 'PB_PushMe', 'BackColor', 'Green'
end
else do
call VRSet 'PB_PushMe', 'BackColor', 'Red'
end
return
```

# Radio buttons


**Figure 42** A radio button

Radio buttons are used to present multiple related options from which only one can be specified. Clicking one radio button selects that object and clears the previously selected radio button. A radio button is set when a solid circle is displayed in the circle to the left of the button's caption.

There are two ways to press a radio button:

oClick on the radio button using mouse button 1.

oIf the radio button has the focus, press the **SpaceBar**.

RadioButton objects can be grouped so that each group operates independently of the other. That is, clicking a radio button within one group only affects other radio buttons within the same group.

## Setting the caption

To change the caption for a radio button, set the Caption property. The button will automatically enlarge if the caption is larger than the button, and the AutoSize property is 1. If the caption is larger than the button, but AutoSize is 0, then the button will not resize and the caption will be clipped.

## Setting and getting the button state

To set a radio button, set its Set property to 1. All other radio buttons in the same group box or window will be reset automatically. You can set a radio button both at design time, and at run time.

To find out if a radio button is set, use VRGet to retrieve the Set property . If it is 1, then the button is set.

### Using the Click event

While it is not necessary to process the Click event for radio buttons, you may want to do so if some object on the window should be updated immediately as a result of the selection.

For example, suppose a window contains two radio buttons (named RB_Red and RB _Green) and a line of descriptive text (named DT_Sample). The color of the descriptive text is to be red if the RB_Red radio button is set, and green if RB_Green is set. To do this, include a click event routine for each radio button. Each routine updates the color of the descriptive text.

The click event routines for the push buttons would be:

```
RB_Red_Click:
call VRSet 'DT_Sample', 'ForeColor', 'Red'
return

RB_Green_Click:
call VRSet 'DT_Sample', 'ForeColor', 'Green'
return
```

## Check boxes

**Figure 43** A check box

Check boxes are used to present multiple options from which one or more options can be selected independently of the other options available. Clicking one check box has no automatic effect on any other check box. Each click on a check box toggles its state between set and clear. A check box is set when a check mark is displayed in the box to the left of the object's caption.

There are two ways to press a check box:

oClick on the check box using mouse button 1.

oIf the check box has the focus, press the **SpaceBar**.

### Setting the caption

To change the caption for a check box, set the Caption property. The button will automatically enlarge if the caption is larger than the button, and the AutoSize property is 1. If the caption is larger than the button, but AutoSize is 0, then the button will not resize and the caption will be clipped.

### Setting and clearing a check box

To set a check box, set its Set property to 1. You can set a check box both at design time and at run time. To find out if a check box is set, use VRGet to retrieve the Set property. If it is 1, then the check box is set.

To clear a check box, set its Set property to 0.

### Using the Click event

While it is not necessary to process the Click event for check boxes, you may want to do so if some object on the window should be updated immediately as a result of the selection.

## Descriptive text

**Figure 44** Descriptive text

DescriptiveText objects are typically used to display titles, prompts, and other static text in a window. This object can show one or more lines of information. Unlike the other text objects, you cannot select or copy descriptive text . Since there is no user interaction with DescriptiveText objects, there are no events associated with them.

### Setting the text

To change the text for a DescriptiveText object, set the Caption property . If the text is wider than the object, one of two things will happen, depending on the value of the AutoSize property:

oIf AutoSize is 0, the text will word wrap at the right edge of the object . You may have to increase the object's height to make the entire caption visible.

oIf AutoSize is 1, the object will make itself wider so that the caption will all fit on one line. The height of the object will not be changed.

The Caption property can be changed both at design time and at run time. For example, changing the text at run time allows you to create a window that monitors and displays changing information.

## Entry field

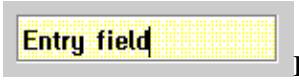

**Figure 45** An entry field

Use an entry field when you want the user to type in a single line of text . If you want multiple lines of text, use a multiline entry field.

At design time, you can set an entry field's Value property to define text that will appear in the entry field when the application is run. Usually you enter data in an entry field at run time. You can enter and edit text using the normal cut, copy, and paste techniques. You can retrieve and change the value entered.

If you want to allow the user to enter more text than the width of the entry field, you should set the AutoScroll property at design time. When you enter text at run time, the field will scroll automatically allowing you to enter or read all of the text.

You can mask entered text to support input of private data such as passwords . When you type text in a masked entry field an asterisk appears for each character typed.

You can define an entry field to be read-only. In this case, you can view but not modify the information. The entry field will be similar to descriptive text except that text selections can be copied.

## Setting and getting the value

To change the value of an entry field, set the Value property. An entry field never changes size to accommodate its value. Use VRGet to retrieve the contents of its Value property.

The length of the value is limited by the TextLimit property. If you want the entry field to contain more text, then set the TextLimit to a larger number ( the default is 255). A text limit of -1 means 'unlimited'.

## Masking the value

There may be times when you want the user to enter some text without it being displayed on the screen. If the user types a password, for example, you may want to hide it. To do this, set the Masked property of the entry field to 1. A mask character appears for each character that you type. You can still set and get the value.

You must set the Masked property at design time. Setting it at run time has no effect.

## Write-protecting an entry field

To write-protect an entry field, set its ReadOnly property to 1. When write-protected, you cannot change the entry field value, although it is still possible to copy its value and paste it elsewhere.

## Using cut, copy, and paste and delete

Entry fields support the usual OS/2 key sequences to cut, copy, paste, and delete text. The keys are listed in the following table.

Cut **Shift+Delete**

Copy **Ctrl+Insert**

Paste **Shift+Insert**

Delete **Delete**

If the entry field has ReadOnly set to 1, only the copy operation is allowed .

### Using default and cancel push buttons

If an entry field is on the same window as a default push button, the button 's Click event is called whenever the **Enter** key is pressed while the entry field has the focus. After, the focus will be left on the push button.

Similarly, if there is a cancel button on the window, its Click event will be called when the **Esc** key is pressed while the entry field has focus.

## Multi line entry field

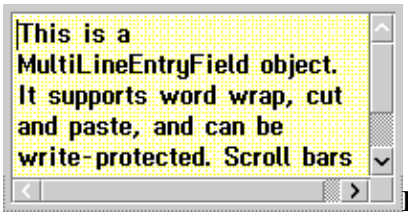


**Figure 46** A multiline entry field

Use a MultiLineEntryField (MLE) when you want the user to enter several lines of text. If you want to enter single lines, you should use an entry field. At run time, you can enter and edit text using the normal cut, copy, and paste techniques.

MLEs can be write-protected in which case you can view but not modify the information. However, you can copy text selections.

MultiLineEntryFields show multiple lines of text. You can scroll the text horizontally and vertically if it cannot be viewed entirely within the object. MLEs also support an option to word wrap if horizontal scrolling is not enabled.

### Adding text to an MLE

To change the value of an MLE, set its Value property. Use VRGet to retrieve the contents of its Value property.

If you want to add text as several lines rather than a single line, separate the lines by a carriage return-line feed pair. The following code adds three lines with line breaks to an MLE:

```
CR_LF = '0d0a'x
value = 'This little piggy went to market' || CR_LF || ,
'This little piggy stayed home' || CR_LF || ,
```

```
'This little piggy programmed all through the night'
call VRSet 'MLE_1', 'Value', value
```

**Note:** A comma at the end of a line is the REXX line continuation character.

The length of the value is limited by the TextLimit property. When you create an MLE, the text limit is set to -1 which means unlimited. If you want the MLE to limit the amount of text, set the TextLimit to the appropriate number.

## Using scroll bars and word wrap

If you add a line of text that is longer than the MLE, it will be word wrapped by default. If you want to preserve the line breaks in the text, you can turn word wrap off by setting the WordWrap property to 0. By default, word wrap is enabled.

While you can always scroll through the MLE text using the cursor keys, you can also use scroll bars to make the task easier. When you create an MLE, the vertical scroll bar property (VertScroll) is set. To add a horizontal scroll bar, set the HorizScroll property of the MLE. Note that if word wrap is enabled, you do not need a horizontal scroll bar. The WordWrap property is set when you create an MLE.

You must set the HorizScroll and VertScroll properties at design time. Setting them at run time has no effect.

## Write-protecting an MLE

To write-protect an MLE, set its ReadOnly property to 1. When it is write -protected, you cannot change the MLE contents at run time. You can, however, copy the text value and paste it elsewhere.

## Using cut, copy, and paste and delete

The MLE object supports the usual OS/2 key sequences to cut, copy, paste, and delete text. The keys are listed in the following table.

Cut **Shift+Delete**

Copy **Ctrl+Insert**

Paste **Shift+Insert**

Delete **Delete**

If the MLE has ReadOnly set to 1, only the copy operation is allowed.

### Using MLEs with a cancel button

If an MLE is on the same window as a cancel push button, the button's Click event is called whenever the **Esc** key is pressed while the entry field has the focus. After, the focus will be left on the push button.

There is no special interaction between an MLE and a default push button.

## Lists


**Figure 47** A ListBox, ComboBox, and DropDownComboBox

VX-REXX provides three list objects: ListBox, ComboBox, and DropDownComboBox. In general, these objects present lists of items from which the user can choose. The list can be sized to show a set number of items. If there are more items than can be displayed, a scroll bar is automatically attached to the list. Selected entries are highlighted.

Use list boxes to display a list of options from which the user can select one or more items. Generally, you use a list box with a push button which, when clicked, performs an action on selected items.

A combo box is a combination of a list box and an entry field. Use a combo box when you want to either select from a list or type in a new value. When you select an item from the list, its text is copied to the entry field part of the combo box, replacing the current value.

Drop down combo boxes are similar to combo boxes except that the list portion of the object is not displayed until requested. Normally, only the entry field part and the drop down arrow are visible. Clicking the arrow indicator makes the list portion visible (but only if the height of the object is larger than the height of a single line of text). Once you select an item from the list, or type in a new value, the drop down list disappears. Drop down combo boxes can be made read-only so that you can only select one of the list items.

### Setting the sort order

You can specify that a list should be sorted automatically when items are added to the list. If you want a list to be sorted, set the Sort property to **Ascending** or **Descending**. If you set Sort to **None**, then the list will be unsorted.

You must set the sort order at design time. Setting the Sort property at run time will have no effect.

## Adding items to the list

You can add items to a list at both design time and at run time. To add items at design time, set the InitialList property. The property is made up of a separator character followed by list items, separated by the separator character. For example, to set the list to be 'Red', 'Green', and 'Blue', set InitialList to ',Red,Green,Blue'. The leading comma indicates that list items are separated by commas.

To add single items at run time, use the AddString method. The following statement adds the item 'White' to list box LB_1 as the third item:

```
call VRMethod 'LB_1', 'AddString', 'White', 3
```

If you omit the position argument, the item is added to the end of the list . The position is ignored in a sorted list, the new item is always added according to the sort order.

To add many items at run time, use the AddStringList method. For example, to add 'Red', 'Green', and 'Blue' to list box LB_1, you would write code similar to:

```
items.0 = 3
items.1 = 'Red'
items.2 = 'Green'
items.3 = 'Blue'
call VRMethod 'LB_1', 'AddStringList', 'items.'
```

Since no position parameter was given, the items will be added to the end of the list.

## Selecting and deselecting list items

You have control over selected items only in list boxes. In combo boxes or drop down combo boxes the selected item is automatically entered into the entry field part. The contents of the entry field are available as the Value property . The index of the selected item is available as the Selected property.

You can configure a list box to allow either single or multiple selection with the MultiSelect property. If MultiSelect is 1, then the user can select more than one item in the list. The MultiSelect property can be set at design time or at run time.

For single selection list boxes, the current selection is reflected by the Selected and SelectedString properties. To select an item at run time, set the Selected property to the item number. For example, if you want to select the third item, set Selected to 3. Similarly, if you want to find out which item is selected, use VRGet to retrieve the value of the Selected property. If you want the text of the selected item, retrieve the value of the SelectedString property.

For multiple selection list boxes, you can use the GetSelectedList and SetSelectedList methods to select and deselect list items.

## Removing list items

To remove a list item, use the Delete, DeleteString or DeleteList methods . To remove all list items, use the Clear method.

To restore a list to its initial state, use the Reset method.

List boxes, combo boxes, and drop down combo boxes support the Clear, Reset, Delete, DeleteString and DeleteList methods.

## Write-protecting drop down combo boxes

You can make a drop down combo box read-only to prevent you from typing a value at run time. You can only select an item from the list. In this case, the drop down combo box acts more like a list box which only allows a single selection. You may want to use the read-only drop down combo box instead of a list box to save space in your window since the list appears only when you want to make a selection.

## Per-item data

User-defined strings can be added on a per-item basis to list boxes, combo boxes and drop down combo boxes. See the AddString, AddStringList, GetString and GetStringList methods.

## Sizing drop down combo boxes

The height of a drop down combo box is the combined height of the list box part and the entry field part that make up the DDCB. If the height is too small, the list box part will not appear when the drop down arrow is pressed.

## Using the Click event

List boxes, combo boxes, and drop down combo boxes generate a Click event when you select an item. The event does not occur for each actual mouse click, only for each new selection, including those done via the keyboard.

When handling a click event for a combo box or drop down combo box, use the Selected and SelectedString properties to find out which item was selected, as the Value property is not updated immediately.

## Using the Change event

When the contents of the entry field portion change in a combo box or drop down combo box, a Change event occurs. In the event routine, use the Value property to get the new value of the entry field portion.

### Using the DoubleClick event

List boxes and combo boxes generate a DoubleClick event when you double click on a list item or press the **Enter** key. The DropDownComboBox object does not have a DoubleClick event; double clicking in that object activates the default push button.

### Using lists with default and cancel push buttons

If a combo box is on the same window as a cancel push button, the button 's Click event is called whenever the **Esc** key is pressed while the combo box has the focus. After, the focus will be left on the push button. Drop down combo boxes also have this behavior.

Pressing **Enter** in a drop down combo box causes the Click routine of a default button to be called. However, there is no special interaction between combo boxes and default push buttons.

## Spin buttons

**Figure 48** A spin button

The SpinButton object lets the user choose one from a set of choices, or to type values directly into an entry field. Only one item from the spin button 's list of items is displayed at a time, so the items should always be in a logical sequence, like a range of numbers or an alphabetized list of names, or the days of the week. If the items do not have a logical sequence, use a ComboBox or DropDownComboBox, both of which provide a list of items and an entry field, but allow the viewing of several items at a time.

### Using spin buttons with numbers

Using spin buttons to let the user select a number is very easy, since VX- REXX already knows the sequence of integers. To set the spin button to cycle through a set of numbers, set its first list item to a string of the form

**<***Min* **to** *Max* **by** *Step***>**

You can do this at design time by setting the InitialList property, or at run time using the SetStringList method. *Min* is the minimum numeric value, *Max* is the maximum numeric value, and *Step* is the increment between items. For example , **<10 to 50 by 10>** would produce the list 10,20,30,40,50. If *Min* is greater than *Max*, the *Min* and *Max* parameters are reversed. If *Step* is negative, it is made positive.

### Using spin buttons with a list of items

Presenting non-numeric list items with a spin button involves a similar procedure to that used for making list boxes. Type the list of items into the spin button's InitialList property, remembering that the first character in the string is used as a separator character. Use that character to separate one item from the next. The separator

character cannot occur in any of the items. For example, the InitialList property for a spin button to display the days of the week may look like:

```
;Saturday;Friday;Thursday;Wednesday;Tuesday;Monday;Sunday
```

Or you could replace the semicolon with any other character that does not appear in the items themselves.

You can also set the list of items at run time with the SetStringList method . This method works like the GetStringList method in reverse. You supply a stem variable containing the items, and they become the spin button's list of items, replacing whatever the list used to be.

The CUA guidelines suggest that if a spin button displays only numeric values , that the up arrow causes the next higher value to be displayed. This is the behavior of the VX-REXX spin button. The guidelines also suggest that if the spin button displays a list of ordered strings, that pressing the down arrow should display the next string. To get this effect, you should list values in the InitialList property in reverse alphabetical order.

### Using the ReadOnly and NumericOnly properties

By default, a spin button will not allow the user to type a value directly into its entry field. If you set the ReadOnly property to 0, the user will be able to enter values directly into the entry field. If you set the NumericOnly property to 1, only numbers can be typed in.

### Using default and cancel push buttons

If a spin button is on the same window as a default push button, the push button's Click event is called whenever the **Enter** key is pressed while the spin button has the focus. After, the focus will be left on the push button.

Similarly, if there is a cancel button on the window, its Click event will be called when the **Esc** key is pressed while the spin button has focus.

## Pictures

VX-REXX provides three picture objects: PictureBox, ImagePushButton and ImageRadioButton. These objects are used to display pictures (icons or bitmaps).

A picture box is a rectangular area for displaying a picture. A border can be drawn around the picture.

An image push button is a push button that displays a picture instead of text . When the user clicks on the image push button, the image it displays is briefly inverted.

An image radio button is a radio button that displays a picture instead of text. When the image radio button is set, its image is inverted. As with radio buttons, only a single radio button in a group (see below) can be set at a time . The VX-REXX tool palette is a collection of image radio buttons.

### Setting the picture path

The picture displayed by a picture object is set using the PicturePath property. The PicturePath can be set to the name of a bitmap or icon file, or the resource identifier of a bitmap or icon resource. See the 'Bitmaps, icons and resources' chapter for more information.

### Creating 3-D effects

You can create 3-D effects for use with the image push button and image radio button objects by including a border as part of the icon or bitmap itself. For example, the image radio buttons in the VX-REXX tool palette all use icons with a white border on the top and left sides, a dark grey border on the bottom and right sides, and a medium grey interior.

## Groups

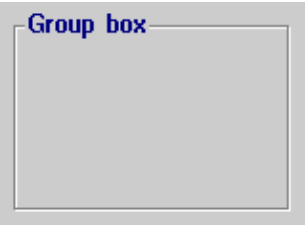


**Figure 49** A group box

Group boxes and windows are containers for objects. All objects added to a window belong to one group. If you need more than one group in a window then you must use a GroupBox object. You should use group boxes for these purposes:

oTo group radio buttons and image radio buttons

All radio buttons (or image radio buttons) within a given group box are mutually exclusive. If you want to put more than one set of radio buttons on a window, you should put each set in its own group box.

Note that all radio buttons which are not in any group box are considered to be in a group. If you are designing a window which uses only one set of radio buttons, you do not need to put them in a group box, although you can for aesthetic reasons.

oTo emphasize the relationship among a group of objects

A group box usually implies that the objects it contains have a common purpose. You can put any type of object into a group box -- even other group boxes .

### Setting a group box caption

The group box caption is displayed on the top line of the group box near the upper left corner. When you create a group box, its caption is the null string . To change the caption, set the Caption property to the new caption. If the caption is longer than the width of the group box, only the leftmost portion of the caption will be visible.

### Adding objects to a group

Objects are added to a group in one of three ways:

1.Selecting an object from the tool palette and then clicking and dragging in the group box.

2.Cutting or copying an object from elsewhere and pasting it into the group box.

3.When the group box is created, any objects it entirely overlaps are automatically added to the group box.

### Removing objects from a group

Objects can only be removed from a group by cutting them from the group box and pasting them elsewhere.

### Disabling objects in a group

When you disable a group box or a window, all of the objects it contains are also disabled, though they will not be grayed out. Similarly, enabling a group box or window enables all of its child objects. However, if a child object 's Enabled property is 0, then the object is always disabled, regardless of whether its parent is enabled or not.

## Notebooks

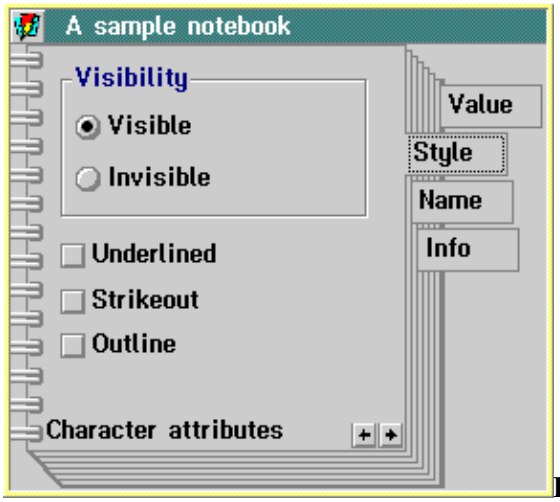


**Figure 50** A notebook

Use notebooks to present information to the user using a book metaphor. The notebook has a number of pages, on which you can place other objects. Pages can have tabs which let the user turn the pages quickly. Notebooks are often used to present settings to the user; OS/2 does this for Workplace Shell objects, and VX-REXX does this for changing object properties at design time.

The parts of a notebook are listed below:

binding The binding is the spine of the notebook. It can be shown either as solid or spiral. The notebook in Figure 50 has a spiral binding.

back pages The pages under the top page make up the back pages. They are drawn along the right and bottom edges in Figure 50.

page buttons The arrows in the corner where the recessed pages meet select the previous and next page.

page window The contents of each page is made up of a page window. A page window is an ordinary VX-REXX window that is used like a secondary window.

status text The text near the bottom edge of the page in Figure 50 is called status text. Each page can have its own status text.

tabs A tab is an extension to a page that the user can click on to select that page. There are two types of tabs: major and minor. Major tabs are always drawn opposite the binding. Minor tabs are shown along the other edge with back pages.

In Figure 50, the words **Value**, **Style**, **Name**, and **Info** appear on major tabs; these words are the tab text. If there were pages with minor tabs, the tabs would appear along the bottom.

When a user clicks on a tab, that page is brought to the front.

tab scroll buttons If the notebook does not have room to display all of the tabs, it creates tab scroll buttons that the user can press to scroll through the list of tabs.

## Changing the notebook binding

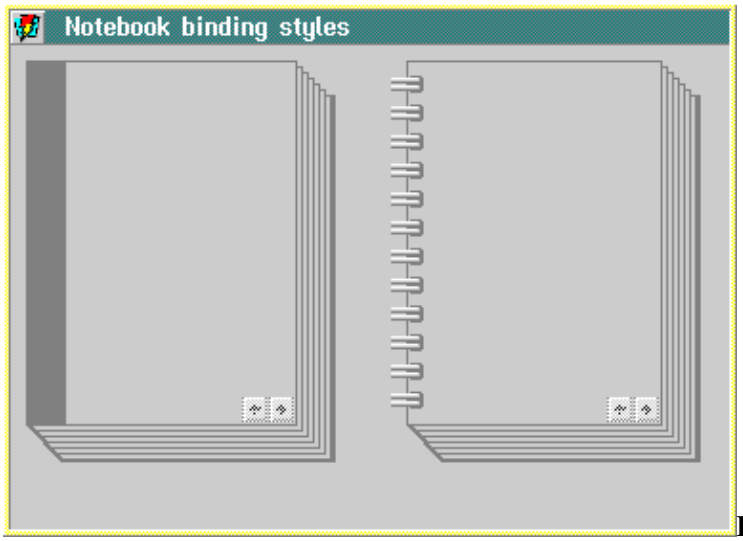


**Figure 51** Notebooks with solid and spiral binding

You can change the notebook binding by setting the Binding property to either **Spiral** or **Solid**. By default, notebooks have a spiral binding. The two styles are shown in Figure 51. The notebook on the left has solid binding, while the one on the right has spiral binding.

This property can be set at design time using the property notebook for the notebook object, and at run time using the VRSet and VRGet functions.

## Changing the tab shape

The appearance of tabs is determined by the TabShape property which can be set to **Square**, **Rounded**, or **Polygonal**. The notebook in Figure 50 has square tabs . By default, pages have square tabs.

This property can be set at design time using the property notebook for the notebook object, and at run time using VRSet. For example, the following statement causes notebook 'NB_1' to display rounded tabs:

```
call VRSet 'NB_1', 'TabShape', 'Rounded'
```

## Using the MajorTabPos and BackPages properties

The BackPages property controls the orientation of the back pages. You can set it at design time and at run time to any of: **TopLeft**, **TopRight**, **BottomLeft**, or **BottomRight**. The BackPages property determines the allowed values for the MajorTabPos property. MajorTabPos can only be one of the two values that make up the BackPages property. For example, if BackPages is **BottomRight**, then MajorTabPos can be either **Bottom** or **Right**.

The binding is always drawn opposite the major tabs. So if MajorTabPos is **Bottom**, then the binding will be drawn along the top edge of the notebook.

Minor tabs are always placed on the side with recessed pages that does not have major tabs. For example, in Figure 51, the minor tabs would appear along the bottom.

The BackPages and MajorTabPos properties can be set at design time using the property notebook for the notebook object, and at run time using VRSet. For example, the following statements put back pages at the bottom right, major tabs along the bottom, and the binding along the top.

```
call VRSet 'NB_1', 'BackPages', 'BottomRight'
call VRSet 'NB_1', 'MajorTabPos', Bottom'
```

## Adding pages at design time

Notebook pages are made up of secondary windows. These windows can be automatically loaded and added to a notebook by listing their names in the notebook's InitialPageList property.

To create a secondary window, select **New Window** from the window list. Add objects to this new window, and write event routines for the objects as you normally would. Then, add the name of the window to the InitialPageList property.

The InitialPageList property is made up of a number of page descriptors, separated by a delimiter which must be the first character of the string. The page descriptors have the following syntax:

```
windowName [ ( + | - ) tabText ]
```

where *windowName* is the name of the window, and *tabText* is the tab text for that page. If you omit the tab text, the page will have no tab. See the description of the InitialPageList property for more information.

The initial page list is processed from left to right. The first window becomes page 1, the second becomes page 2, and so on. Many of the notebook methods take a page number as a parameter. The page number is a whole number from 1 to *n* where *n* is the number of pages in the notebook. There is no fixed limit on the number of pages that a notebook can hold.

Notebooks process the initial page list property in the main event loop of the window file. This means that the pages will not be loaded when the Init routine is called. If you need to perform some initialization on the notebook pages , you should put your initialization code for a given page in that window's Create event routine. Your code will be called when the window is loaded.

For an example of how to use the InitialPageList property, recall the notebook in Figure 50 which has four pages. The secondary windows that

are the pages of the notebooks are named **SW_1, SW_2, SW_3,** and **SW_4.** The InitialPageList string to add these pages with their tab text would be:

```
;SW_1+Value;SW_2+Style;SW_3+Name;SW_4+Info
```

Note that secondary windows are displayed slightly differently when they are added to a notebook. You should be aware of the following points:

oSecondary windows are positioned in the upper left corner of the page. If the window is larger than the page, only the top left portion of the window will be visible. The user can make the rest of the window visible by making the notebook larger.

oPage windows have no frame, so the BorderType, MaximizeButton, MinimizeButton, HideButton, SystemMenu and WindowState properties have no effect.

oThe WindowMode property of the window is ignored.

Menus, however, will be displayed.

## Setting the page tab text

To change the tab text for a page, use the SetTabText method. The notebook will automatically resize the tabs to fit the longest tab text. The GetTabText method returns the tab text for a given page.

## Setting the page status text

To change the status text for a page, use the SetStatusText method. The GetStatusText method returns the status text for a given page. The status text can be set only at run time.

## Turning to a given page

To set the current page number, use VRSet to change the Selected property . Set the property to 1 to turn to the first page. To turn to the last page, use VRGet to retrieve the number of pages in the Count property, then turn to that page.

While Selected can be set and retrieved at run time, the Count property is read-only at run time.

## Adding pages at run time

Notebooks automatically insert all pages listed in the InitialPageList property when they are created. To add more pages at run time, use either the InsertBlankPage method or the InsertPage method. Use InsertBlankPage if the page window has not been loaded yet; use InsertPage if the window is already loaded.

InsertBlankPage reserves a page for a window in the notebook. When that page is selected, the notebook will automatically load the page window (see the PageLoad section in the *Reference*)

### Preloading notebook pages

By default, the notebook loads a page window when the page is initially selected. This scheme results in better performance than loading all of the page windows when the notebook is created. If your program requires that all page windows be loaded on startup, set the PreloadPages property to 1. By default, PreloadPages is 0.

### Loading pages under program control

By default, notebooks load page windows automatically. If the application needs to perform the loading itself, it must define a PageLoad event for the notebook . When a page is selected for the first time, this event is generated. The event handler must call VRLoad to load the window, then invoke the SetPageWindow method to associate the window's internal name with the notebook page.

The following PageLoad event is equivalent to the notebook's default behavior:

```
NB_1_PageLoad:
name = VRInfo( 'WindowName' )
page = VRGet( 'NB_1', 'Selected' )
w = VRLoad( 'NB_1', VRWindowPath(), name )
if w \= '' then do
call VRMethod 'NB_1', 'SetPageWindow', page, w
end
return
```

**Note:** Do not use VRLoadSecondary to load notebook page windows. VRLoadSecondary does not make the loaded window a child of the notebook. It also makes the window visible as soon as it is loaded, which may cause the window to flash as it is added to the notebook.

### Removing pages at run time

Pages can be deleted using the DeletePage method. (This does not destroy the page window.) When a notebook is destroyed, it automatically deletes and destroys all of its pages.

## Containers

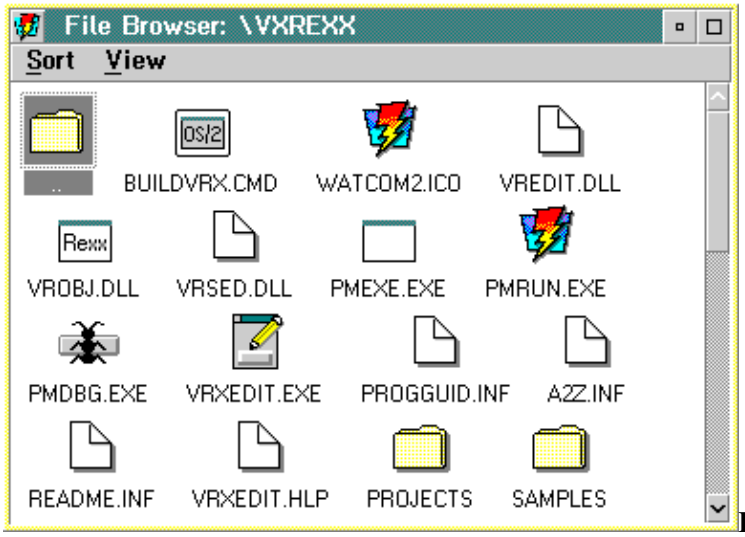


**Figure 52** A container

A container shows a view of a set of records. The view may consist of icons, bitmaps, text, or a combination of these. All records have attributes such as an icon, caption, and its position within the container.

Records have a hierarchical structure. A record may be the parent of a number of other records. This parent-child relationship can be shown using the container tree views.

Containers support drag and drop interactions. A user can do the following, among others:

oDrag files onto containers
oDrag records onto the Workplace Shell Shredder
oDrag records onto Workplace Shell printers

OS/2 uses containers in the Workplace Shell; the desktop and folder objects are containers. The icon, tree, and detail views used by the Workplace Shell are fundamental container properties that you can use in your VX-REXX programs.

## Setting the view

The type of view that a container displays is set using the View property . The following sections describe each view.

**Icon view**

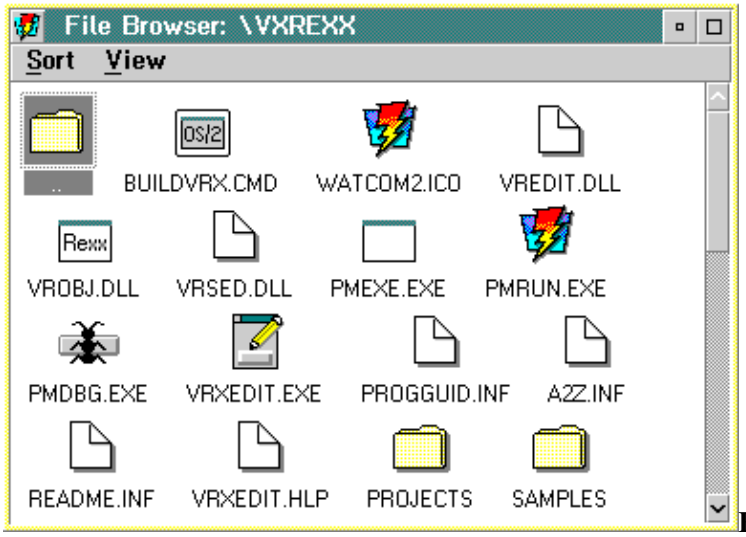


**Figure 53** A container in icon view

In icon view, the icon and caption for each record are shown together. Icons can be positioned anywhere in the container workspace. If there is not enough room to display all the records at once, the container adds horizontal and vertical scroll bars where necessary.

**Name view**

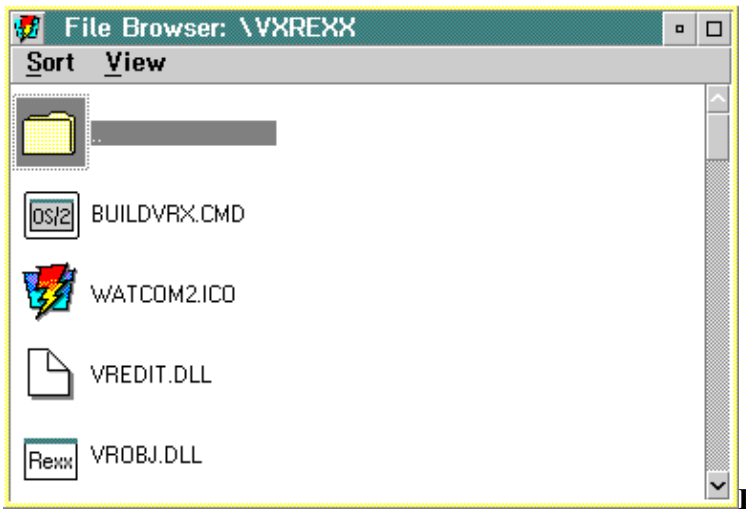


**Figure 54** A container in name view

In name view, the icon and caption for each record are shown side by side . Records are listed in a single column, unless the Flowed property is set, in which case the column wraps horizontally.

**Text view**

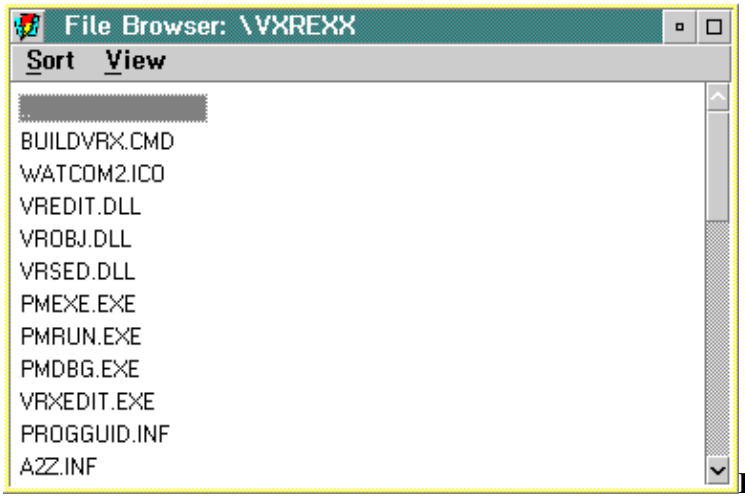


**Figure 55** A container in text view

In text view, only the record captions are displayed. Records are listed in a single column, unless the Flowed property is set, in which case the column wraps horizontally.

**Detail view**

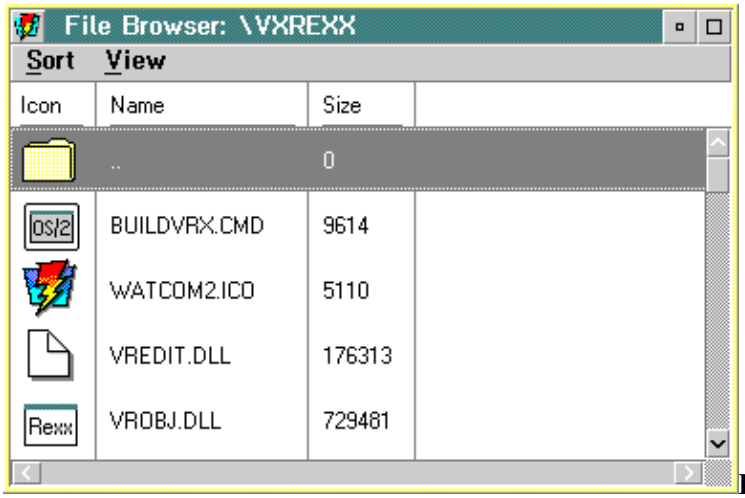


**Figure 56** A container in detail view

The detail view presents the records in tabular form. The columns are made of fields which the REXX program creates, while each row contains information for a given record. Fields can contain icons, text, numbers, dates, and times .

The record icon and caption are not shown in the detail view.

The detail view can be optionally split into left and right halves. The two halves share the same vertical scroll bar, but can be scrolled horizontally independently. See the SplitBarLeft and LastSplitField properties for more information .

**Tree views**

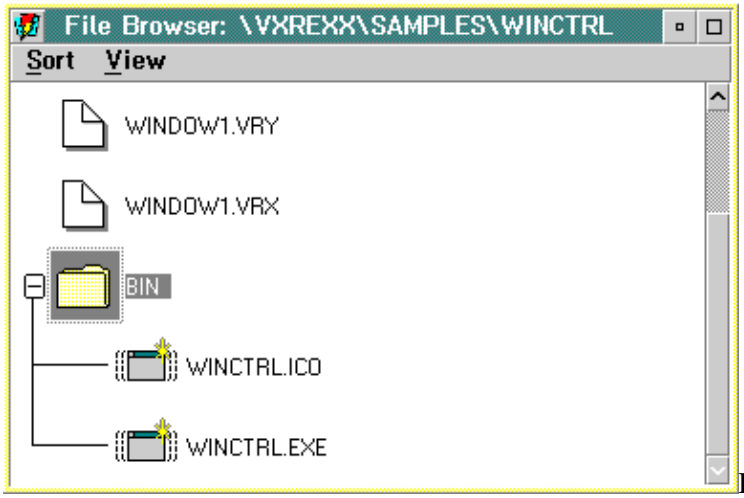


**Figure 57** A container in icon tree view

Records can be organized in a hierarchical structure with parent and child records. Child records are displayed in the following views: icon tree, name tree, and text tree. Child records are never displayed in icon, name, text, or detail views.

Figure 57 shows a container whose View property is set to **IconTree**. The BIN record has two child records: WINCTRL.ICO and WINCTRL.EXE.

In the icon tree or text tree views, a graphic indicator appears to the left of each parent record to indicate that the record has children. If the children are not displayed, the parent record is said to be *collapsed*. When the children are showing, the parent is *expanded*. No indicator appears in name tree view , although clicking on the parent record makes the children visible.

A line connects the child and parent records. This line can be suppressed by setting the ShowTreeLine property to 0. The thickness of the line can is set by the TreeLine property, while the amount of indentation used to display child records is set using the TreeIndent property.

## Creating detail view fields

In the detail view, the record caption and icon are not displayed. Instead the container shows records in a tabular form. Each row contains information associated with a given record. The information is broken up horizontally into a number of fields.

Detail view fields are only visible in the detail view. If a container is never used in detail view, there is no need to define fields or to set field data . You may want to do so however, in order to store user defined information about records.

It is up to the REXX program to create each field and to set the field data for each record. Fields are created using the AddField method. When you add a field, you must declare both the field caption and the type of data that the field will hold. For example:

```
nameField = VRMethod( 'CN_1', 'AddField', 'String', 'Name', 'NameField' )
```

adds a field with the caption 'Name'; the field holds strings. The last parameter **NameField** is the symbolic name for the new field. The _AddField method returns a *field handle* which uniquely identifies the field. The method that sets field data for records takes the field handle as a parameter. You can also use the symbolic name wherever you use a field handle.

**Note:** You must define all fields before adding any records to the container.

Fields have a number of attributes which can be set using the SetFieldAttr method. The REXX program can configure a field's visibility, text justification, and read/write status. For example, the following statement makes the name field created above read only.

```
call VRMethod 'CN_1', 'SetFieldAttr', nameField, 'ReadOnly', 1
```

The default field attribute values are listed in the SetFieldAttr section in the reference guide.

## Record emphasis

Several types of record emphasis are available. Each emphasis type may be set using SetRecordAttr and retrieved using GetRecordAttr:

Visible If set, the record will be shown.

Cursored If set, a thin dotted line is drawn around the record. The cursored record is the record with input focus. The cursor keys can be used to move to cursor to another record. Pressing the space bar will select the cursored record.

Selected If set, a solid background is used for the record. Selection is done using the space bar (to select the cursored record) or by clicking on the record with mouse button 1. Record selection is influenced by the container's _MultiSelect and ExtendSelect properties. If both properties are 0, only a single record can be selected at a time. If MultiSelect is 1, any number of records can be selected and clicking on the record either selects it (if it was deselected) or deselects it (if it was selected). If MultiSelect is 0 but ExtendSelect is 1, clicking on a record will select it and deselect the rest unless the **Ctrl** is down, in which case the click selects or deselects the record without affecting the other selected records. Swipe selection is also enabled if either MultiSelect or _ExtendSelect is 1.

InUse If set, a hatched emphasis is used for the record. This is the same emphasis used by Workplace Shell objects to indicate a view has been opened. This emphasis can only be set by the program.

Source If set, a heavy dotted line surrounds the record. This emphasis is set by VX-REXX when a pop-up menu is displayed in a container and is used to indicate which records are to be acted on by the commands in the pop-up menu. If the record you clicked on to open the pop-up (see the ContextMenu event) is selected, source emphasis will be set on all selected records. Otherwise only the record you clicked on has the source emphasis set.

The GetRecordList method is used to obtain a list of records in a container that have a given emphasis.

Pull-down menus should only operate on the selected records in a container . Pop-up menus should only operate on the records with source emphasis. The **SourceOrSelected** parameter may be passed to GetRecordList to obtain the correct list of records . Also, if an item is selected from a pop-up menu, it is your project's responsibility to remove the source emphasis from any records.

## Adding records

Use the AddRecord or AddRecordList methods to add records to a container. When you add a record, you can give its initial position in the container workspace, its caption, and its icon. You can later change any of this information using the SetRecordAttr method.

If you use the AddRecord method, detail view data must be specified later with the SetFieldData method. If you use AddRecordList, you can create the records and initialize the field data all at once. See the AddRecordList method for details.

The AddRecord method returns a record handle which is an identifier for the new record. The record handle is passed to many of the container methods.

You do not have to add records to each particular view. Records are added to the container, and the container displays its records in some particular view .

The following sample code creates a record with the caption **Anaconda**. It uses the icon in file SNAKE.ICO.

```
record = VRMethod( 'CN_1', 'AddRecord', '', 'Last',,
'Anaconda', 'SNAKE.ICO' )
```

The null string in the third parameter above indicates that the record has no parent. If this had been a child record, this parameter would have been the record handle of its parent.

The fourth parameter to VRMethod is **Last**, meaning that the record should be added at the end of the container's record list. In the name, text, and detail views records are listed in the order in which they were added.

The trailing comma on the first line in the above example is the REXX line continuation character.

The following sample shows how to use AddRecordList to add a number of records all at once.

```
info.0 = 3 /* Create 3 records */
info.1 = ';Apple;APPLE.ICO'
info.2 = ';Orange;ORANGE.ICO'
info.3 = ';Grape;GRAPE.ICO'
ok = VRMethod( 'CN_1', 'AddRecordList', , , 'info.' )
```

**Note:** Once you add the first record to a container, you can no longer add any fields.

## Removing records

Use the RemoveRecord method to remove a record. You do not have to remove records from each view because the record belongs to the container, not the individual views. If the AutoPosition property is 1, the container will automatically adjust the other records' positions as a result of the deletion.

## Selecting records

You can control the way records are selected in a container by setting the ExtendSelect and MultiSelect properties. The three selection modes are described below .

**Single selection (ExtendSelect=0, MultiSelect=0)**

You can select one record at a time. Select a record by clicking on it, or by moving the cursor to the record then pressing **Ctrl+Spacebar**.

**Multiple selection (ExtendSelect=0, MultiSelect=1)**

You can select one or more records at a time in any order. Clicking on a record toggles its selected state without deselecting other records.

Multiple selection is not available in any of the tree views.

**Extended selection (ExtendSelect=1, MultiSelect=0)**

You can select one or more records at a time in any order. Clicking on a record automatically deselects all other selected records. To add a record to the selected list, hold the **Ctrl** key while clicking.

Extended selection is not available in any of the tree views.

**Note:** The container ExtendSelect and MultiSelect properties operate differently than those on list boxes and combo boxes.

## Setting detail view information

To set the detail view information, use the SetFieldData method. If you do not set the field data for a given record, the field will be left blank if it is a string, or 0 otherwise.

The following code sets the name field for record **rh** to 'Cattran' :

```
ok = VRMethod( 'CN_1', 'SetFieldData', rh, nameField, 'Cattran' )
```

## Making fields invisible

You cannot remove fields from a container, although it is possible to hide fields, which has the appearance of removing fields.

Sometimes you may want to associate extra information with each record, but you do not want that information displayed. In this case, you can store the information in an invisible field. You can make fields invisible by setting the field attribute **Visible** to 0.

## Positioning records

In icon view, records are positioned according to their Left and Bottom attributes. These two attributes give the record's location relative to the container's workspace. The workspace is a logical area equal to the rectangle that is just large enough to enclose all records in icon view. If the workspace is smaller than the container object, scroll bars appear to let the user scroll through the workspace.

You can change the position of a record using the SetRecordAttr method. The following statement moves the record whose handle is in **rh** 200 twips above the bottom of the workspace, and 500 twips to the right of the left edge:

```
call VRMethod 'CN_1', 'SetRecordAttr', rh, 'Bottom', 200, 'Left', 500
```

In icon view, use the Arrange method to reposition records according to the CUA guidelines. Records are put in a row near the top of the workspace. Once the row is full, a new row is started under the first row, and so on. If the AutoPosition property is 1, then the container automatically positions records in icon view. The records' Left and Bottom attributes are ignored in that case.

In the other views, records are displayed in the order they are added. This order can be changed by sorting the records.

## Sorting records

Container records can be sorted at run time. To sort records, follow these steps:

1.Set the Sort property to indicate the ordering you want.

2.If you are sorting while the detail view is shown, set the DetailSort property to the field handle which contains the sort key.

3.Invoke the SortRecords method.

If the container is not showing the detail view, sorting is done on the records' caption attribute.

If the AutoSort property is 1, then the container automatically sorts records as they are inserted.

## Searching records

Like items in a list, records can be searched for a particular text string . Use the FindRecord method to do this.

## Sharing records between multiple containers

Once you create a record using the AddRecord method, you can then add it to other containers using the AddSharedRecord method. The record is then said to be *shared*. While the record's caption, icon, and field data are shared between all containers, each container keeps track of the record's position separately . For example, if you change the Caption attribute of a shared record in one container, that change will be reflected in all of the containers. However, changing the Left or Bottom attributes in one container does not affect any of the other containers.

You remove shared records in the usual way with the RemoveRecord method. The record will not actually be destroyed until it is removed from all of the containers it has been added to.

## Moving a record from one container to another

You can use AddSharedRecord to move records from one container to another while preserving the detail field information. The following DragDrop event routine illustrates how to do this:

```
CN_1_DragDrop:
srcCtn = VRInfo( 'SourceObject' )
srcRec = VRInfo( 'SourceRecord' )
targetCtn = VRInfo( 'TargetObject' )
left = VRInfo( 'Left' )
bottom = VRInfo( 'Bottom' )

ok = VRMethod( targetCtn, 'AddSharedRecord', srcRec )
if ok then do
ok = VRMethod( targetCtn, 'SetRecordAttr', srcRec,,
'Left', left, 'Bottom', bottom )
ok = VRMethod( srcCtn, 'RemoveRecord', srcRec )
end

return
```

You can also use the AddRecord method to create a new record in the target container, then delete the source record. The disadvantage to this technique is that none of the detail field values are preserved. Also, it is up to the REXX programmer to preserve the record's Caption and Icon

attributes.

## Direct editing of records

A user can modify the records' caption text and field data by clicking mouse button 1 over the text while holding down the **Alt** key. This action opens an editing field over the record in which the user can type new data. A direct editing field can also be opened by invoking the OpenEdit method.

The user finishes a direct editing operation by clicking on another record to accept the changes, or by cancelling the operation. Any of the following actions will cancel the operation and restore the record data to its previous value:

oPress the **Esc** key
oDrag the record being edited
oClick on another record while holding the **Alt** key
oScroll the container

Invoking the CloseEdit method will close the direct editing field without losing the changes.

If the AutoEdit property is 1, the user can tab between fields while direct editing. This feature is available only in detail view.

A BeginEdit event occurs when a direct editing operation starts. If AutoEdit is 0, the event occurs for each field being edited. When AutoEdit is 1, the event occurs only once for each record being edited. Similarly, the EndEdit event occurs when the direct editing operation finishes. If AutoEdit is 0, an EndEdit event occurs for each field. Otherwise, it occurs for the record as a whole.

To prevent direct editing of the record's caption, set the record's ReadOnly attribute to 1. If the container's ReadOnly property is 1, no direct editing is allowed on any record, regardless of the record's ReadOnly attribute . In detail view, direct editing is also inhibited if the field's ReadOnly attribute is 1.

**Note:** Direct editing is supported for string fields only. You cannot direct edit a field of type Date, Icon, or Time.

## Checking if a record is in a container

To check if a record is in a container, use the ValidateRecord method. The following sample code tests if the record whose handle is stored in **record** is currently in container CN_1:

```
if( VRMethod( 'CN_1', 'ValidateRecord', record ) = 1 ) then do
say 'The record is in the container.'
end
```

## Using the container ContextMenu event

When the user causes a ContextMenu event by clicking mouse button 2, it automatically sets the Source attribute for the records and the Source property for the container to indicate the records to which the ContextMenu event applies. If the user does not select a menu item from the pop-up menu, the source emphasis is removed automatically. If the user clicks on a menu item, it is up to the menu item Click routine to reset the Source emphasis.

The container applies the source emphasis according to the following rules :

oIf the user clicks mouse button 2 on an unselected record, only that record will have its Source attribute set to 1.

oIf the user clicks on a selected record, all selected records will have the Source attribute set to 1.

oIf the user does not click on a record, the container's Source property is set to 1. The Source attribute for all records is reset to 0.

For an sample menu click routine, see Using pop-up menus with containers.

## Dragging and dropping records

For a description of drag and drop operations in general, see the 'Drag and drop operations' chapter. The information presented below describes the drag and drop support available to containers, which is similar (but not exactly the same) to the drag and drop support available to other objects.

**Overview**

Users can drag and drop container records in the same way that they drag and drop VX-REXX objects and Workplace Shell objects. In general, VX-REXX supports the following drag/drop operations:

oRecords can be dropped on any other object (including records in other containers) in the same program.

oRecords can be dropped onto containers or records from other programs if the records represent actual OS/2 files (they must have the Filename attribute set).

oRecords can be dropped onto Workplace Shell objects or containers in other programs if the records represent actual OS/2 files (they must have the Filename attribute set).

oRecords can be dropped on the Workplace Shell Shredder and any Workplace Shell printer object.

oOS/2 files and other objects can be dropped on containers and records.

The REXX program can prevent drag/drop operations in the following ways :

oYou can specify the types of things that can be dropped onto a given record by setting the record's Target attribute.

oYou can specify the types of things that can be dropped onto a container by setting the container's DragTarget property.

oRecords cannot be dropped onto a read only container.

oThe user cannot drag records that have the AllowDrag attribute set to 0.

oRecords in a container with the AllowDrag property set to 0 cannot be dragged, regardless of the records' AllowDrag attribute setting.

In all of the cases listed above, it is the responsibility of the REXX program to respond to container events to perform the action requested by the drop operation. For example, if the user drags a record from one container to another, the REXX program must add the record to the target container then delete it from the first container.

**Drag/drop operations**

When the user drops a record, there is an implied action associated with the drop. It is one of: **Copy**, **Link**, or **Move**. The default operation when dragging records between containers in the same VX-REXX program is **Move**. The user can change the operation by holding down the **Shift** and **Ctrl** keys while dragging, as outlined below:

Copy The **Ctrl** key was held down. The dragged icon appears partially transparent.

Link The **Ctrl** and **Shift** keys were held down. A line is drawn between the record and the mouse while dragging to indicate the link operation.

Move The **Shift** key was held down. The dragged icon is drawn normally.

The REXX program can retrieve the operation type by passing **Operation** to VRInfo in the following events: DragDrop, DragFile, and MoveRecord. The REXX program can provide a default operation by invoking the StartDrag method from within a DragStart event.

**Describing eligible drag/drop sources**

You can program the container to define the types of objects that can be dropped onto it. The list of accepted objects is stored in the container's DragTarget property. By default, this property is set to **All**, meaning that container records, Workplace Shell files and other VX-REXX objects can be dropped onto the container. If you want only records to be dropped, set DragTarget to **Records**. If you do not want the user to drop anything on the container, set DragTarget to **None** .

You can also restrict the types of objects that are dropped onto a given record. To do this, set the record's Target attribute using SetRecordAttr .

Note that the container's DragTarget property applies only to the ' white space' of the container, not the records it contains.

**Using the Source attribute in drag/drop operations**

The container automatically sets the Source attribute for dragged records . This forms a visual cue to the user which highlights the records that are about to be dragged. If the user drags an unselected record, then only that

record will have the Source attribute set. (Another way of saying this is that the record has the Source *emphasis*.) If the user drags a selected record, then all selected records get the Source emphasis. Any records that are not involved in the drag/drop operation have the Source attribute reset to 0.

When records are dropped onto a container, the target container receives a DragDrop event for each record that was dropped. It is then up to the DragDrop event to turn off the Source attribute for each dropped record. If it is more appropriate to process all dropped records at once rather than one at a time, the target can use the GetRecordList method to list all records in the source container that have the Source attribute turned on. The DragDrop event can then process all the records at once.

It is up to the program to turn off the source attributes of dragged records .

The following sample DragDrop event causes all records that are dropped onto its container to be deleted, just like the OS/2 Shredder. The routine processes all of the dropped records at once. Note that the target container will receive a DragDrop event for *each* dropped record. However, since we turn the Source attribute off for all of the dropped records when the first event occurs, nothing will happen in the remaining events. The other DragDrop events still occur, but the GetRecordList method will return an empty list of records.

```
CN_1_DragDrop:
container = VRInfo( 'SourceObject' )
call VRMethod container, 'GetRecordList', 'Source', 'reclist.'
do i = 1 to reclist.0
call VRMethod container, 'SetRecordAttr', reclist.i, ,
'Source', 0
call VRMethod container, 'RemoveRecord', reclist.i
end
return
```

If you wanted to purge the remaining DragDrop events from the event queue, you could use the VRFlush function.

**Programming with the MoveRecord event**

The MoveRecord event occurs when the user drags a record to a new position within the same container. The event occurs if the user drags a record onto the container white space, or onto another record. If the record's Target attribute does not include records, no MoveRecord event will be generated. Similarly, if the container's DragTarget property does not include records, dropping a record on the container's white space will not generate an event.

If the container is in a linear view (the View property is set to **Name**, **Text**, or **Detail**), and no MoveRecord event is defined, the dragged records will be automatically moved. If the container is not in a linear view, or if a MoveRecord event has been defined for the container, no reordering occurs; it is up to the REXX program to reorder the records by setting the Previous record attribute using the SetRecordAttr method.

Suppose you wanted the drag and drop operation to reparent the dropped records, similar to dropping files into a folder in the Workplace Shell. The following MoveRecord event code does this by setting the Parent attribute of the dragged record to the target record.

```
CN_1_MoveRecord:
srcRec = VRInfo( 'SourceRecord' )
targetRec = VRInfo( 'TargetRecord' )
if( targetRec \= '' ) then do
call VRMethod 'CN_1', 'SetRecordAttr', srcRec, ,
'Parent', targetRec
end
return
```

The record reordering behavior for linear views is equivalent to the following code:

```
CN_1_MoveRecord:
srcRec = VRInfo( 'SourceRecord' )
targetRec = VRInfo( 'TargetRecord' )
if( targetRec = '' ) then do
targetRec = 'Last'
end
call VRMethod 'CN_1', 'SetRecordAttr', srcRec, ,
'Previous', targetRec
return
```

Note that for the above event routine, dropping a record on the container white space moves the record to the end of the list.

A MoveRecord event is generated for each record that is moved. If you need the entire list of dragged records, you can get it by defining a DragStart event:

```
CN_1_DragStart:
call VRMethod 'CN_1', 'GetRecordList', 'Source', 'Globals.!RecList'
call VRMethod 'CN_1', 'StartDrag'
return
```

The above code uses the GetRecordList method to get the list of records that are about to be dragged. The list is stored in the REXX variable **Gloabls.! RecList**. In the MoveRecord event, you can use the list to manipulate the dragged records. You cannot get the list of dragged records in the MoveRecord event because the container automatically turns off the Source attribute when records are dragged within the same container. Note that the Source attribute is left on when records are dragged from one container to another, or to the Workplace Shell.

**Programming with the DragDrop event**

The DragDrop event is generated when the user drops a record from another container or a Workplace Shell file into a container or one of its records. The target container receives the DragDrop event.

The DragDrop event is similar to the MoveRecord event, but has the following differences:

oA DragDrop event is generated if the items being dropped are not records from the same container. Dragging and dropping records within a single container causes a MoveRecord event, not a DragDrop event.

oNo automatic record reparenting or reordering occurs under any circumstance when a DragDrop event occurs.

It is the responsibility of the REXX program to take some appropriate action in response to a DragDrop event.

The following sample event routine copies or moves a record from the source container to the target container.

```
CN_1_DragDrop:
/* Get information about the drag drop source and target
*/
srcCtn = VRInfo( 'SourceObject' )
srcRec = VRInfo( 'SourceRecord' )
targetCtn = VRInfo( 'TargetObject' )
left = VRInfo( 'Left' )
bottom = VRInfo( 'Bottom' )
operation = VRInfo( 'Operation' )

/* Add the record to the target container
*/
call VRMethod targetCtn, 'AddSharedRecord', srcRec
call VRMethod targetCtn, 'SetRecordAttr', srcRec,,
'Left', left, 'Bottom', bottom

/* If moving the record, remove it from the source container
*/
if( operation = 'Move' ) then do
call VRMethod srcCtn, 'RemoveRecord', srcRec
end
return
```

## Interacting with the Workplace Shell

**Dropping files onto a container**

The user can drop Workplace Shell file icons onto containers or records. When this happens, a DragDrop event is generated. The REXX program must respond to this event in the same way as if a record had been dropped onto the container . The program can get the name of the file by calling VRInfo in the DragDrop routine.

The following DragDrop event routine prints the names of files dropped onto it.

```
CN_1_DragDrop:
filename = VRInfo( 'SourceFile' )
if filename \= '' then do
say 'File' filename 'was dropped onto me.'
end
return
```

You can program the container to accept only certain types of files by setting the DragTarget property of the container. By default, any file can be dropped. If you set the DragTarget property to a file mask (e.g. '* .TXT'), then only files whose names match the mask can be dropped on the container.

You can also restrict the types of files that are dropped onto a given record . To do this, set the record's Target attribute using SetRecordAttr.

**Dropping records onto Workplace Shell objects**

The user can drag records onto Workplace Shell folders or applications as long as the records represent actual OS/2 files. To tell the container about the file a record represents, set the record's Filename attribute to the name of the file including path and drive information. The record icon will change to the one that would be shown in a Workplace Shell folder for that file.

The following sample code adds a record to a container, then sets its Filename attribute to **C:\CONFIG.SYS**. At run time, the user can drop the record onto EPM to view the file. (EPM is the OS/2 Enhanced Editor, and is initially installed in the **Productivity** folder which is in the **OS/2 System** folder.)

```
record = VRMethod( 'CN_1', 'AddRecord',,, 'CONFIG.SYS' )
call VRMethod 'CN_1', 'SetRecordAttr', record, 'Filename', ,
'FileName', 'C:\CONFIG.SYS'
```

When a record is dropped onto a Workplace Shell folder or application, the source container receives a DragFile event. If the REXX program does not define a DragFile event, and the target moves or deletes the file that record represents, then the record is automatically removed from the container. If a DragFile event *is* defined, it is up to the event routine

to remove the record.

The default container action is equivalent to the following DragFile event :

```
CN_1_DragFile:
record = VRInfo( 'SourceRecord' )
file = VRInfo( 'SourceFile' )

if( \VRFileExists( file ) ) then do
call VRMethod 'CN_1', 'RemoveRecord', record
end
return
```

**Displaying file icons**

In an OS/2 folder, files are displayed with a certain icon. You set the icon for a record to the same icon that is displayed for a given file. To do this, set the Icon attribute to the name of the file with path and drive information .

For example, the following code creates a record whose icon is the same for the OS/2 Chess program. Note that the record does not represent the file since the Filename attribute is not set. This means that the user cannot drag the record to Workplace Shell folders or applications.

```
record = VRMethod( 'CN_1', 'AddRecord',,,'Chess',,
'C:\OS2\APPS\OS2CHESS.EXE' )
```

If you want to use a container to display a number of files, use the FillFromDir method to add records based on the contents of a directory. The method adds a record for each file that matches a file mask in the given directory. If no detail view fields have been defined, it creates the same ones that are in the detail view for Workplace Shell folders -- icon, name, size, date, and so on. The fields are created with symbolic names that correspond to the field headings: **FileIcon**, **FileName**, **FileSize**, **FileWriteDate**, **FileWriteTime**, **FileAccessDate**, **FileAccessTime**, **FileCreateDate**, **FileCreateTime**, and **FileFlags**. If you define a field with one of these names before calling the FillFromDir, the method will use your field instead of creating one. The method also sets the record Caption, Icon, and Filename attributes automatically.

The following code fills a container with records for all the executable files in the 'D:\VXREXX' directory:

```
call VRMethod 'CN_1', 'FillFromDir',, 'D:\VXREXX', '*'
```

**Programming with the DragDiscard event**

If the user drops a record on a Workplace Shell Shredder object, the container to which the record belongs is sent a DragDiscard event. The container does not automatically delete the record; the event routine must do this. If no _DragDiscard event is defined, then nothing happens.

The following sample event routine deletes a record when it is dropped on the Shredder.

```
CN_1_DragDiscard:
record = VRInfo( 'Record' )
call VRMethod 'CN_1', 'RemoveRecord', record
return
```

**Programming with the DragPrint event**

If the user drops a record on a Workplace Shell printer object, the container to which the record belongs is sent a DragPrint event. The container does not automatically print the record; the event routine must do this. If no DragPrint event is defined, then nothing happens.

## Programming with the DragStart event

By default, when the user starts to drag a record (or records), the drag operation starts right away -- that is, the user sees the record move with the mouse pointer. Your program can get control between the time the user clicks mouse button 2 and the time the records are actually dragged by defining a DragStart event .

If your program defines a DragStart event, it is up to your program to actually start the drag operation by invoking the StartDrag method. If this method is not used, the records will not be dragged. The default container action is equivalent to the following DragStart event:

```
CN_1_DragStart:
call VRMethod 'CN_1', 'StartDrag'
return
```

StartDrag only causes records with the Source emphasis to be dragged. The _DragStart event can get the list of records that have the source emphasis using the _GetRecordList method. It can then reset the Source attribute if the record should now be dragged, or it can set the Source attribute of some other record so that it will be dragged. All of the Source attribute manipulation must be done before invoking the _StartDrag method.

The DragStart event is also a convenient time to set the Filename attribute for the dragged records. Suppose that the container is full of records that represent information in a database. The intent is for the user to drag one of the icons to EPM in order to view the information. In order to drag a record to EPM, the record must represent an OS/2 file, that is, the Filename attribute must be set . Rather than build and update files for all of the records, you can create a file when the user drags the record. This approach is more efficient because files are created only for the records that the user wants to view.

The following sample shows a DragStart event that creates a file for each dragged record, then sets the Filename attribute for the records. The files created in this example hold some dummy text; in a real program you would put more meaningful data.

```
CN_1_DragStart:
/* Create a file for each record
*/
call VRMethod 'CN_1', 'GetRecordList', 'Source', 'reclist.'
do i = 1 to reclist.0
filename = 'TMP' || i
call Lineout filename, 'Dummy text for record' i
call VRMethod 'CN_1', 'SetRecordAttr', reclist.i, ,
'Filename', filename
end

/* Start the drag operation
*/
call VRMethod 'CN_1', 'StartDrag'
return
```

## Sliders

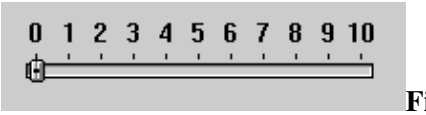


**Figure 58** A slider

A Slider object is used to let the user select a value from a range of values . The slider also shows the current value relative to the entire range of values . The slider can also be used as a status display, to indicate progress during a long operation such as copying a large number of files.

Interacting with a slider is much like interacting with a scroll bar. The user moves the *slider arm* by dragging it, or by clicking on the *slider shaft* or slider buttons.

### Setting up the slider ticks

To specify the number of ticks on the slider, set the Ticks property to the number of ticks you want. Labels for the ticks are taken from the TickList property. If there are fewer ticks than labels in the TickList, then the rest of the labels are ignored.

To leave a tick unlabeled, add an empty item in the tick list. For example, if you wanted to have 3 ticks where the first is labeled **0**, the last is labeled **100**, and the middle tick is unlabeled, you would set Ticks to 3 and TickList to **;0;;100**.

### Setting and getting the slider value

There are two properties that set the slider value: Percentile and TickIndex. Percentile sets the slider value as a percentage (in whole numbers) of the distance between the home position and the opposite end. TickIndex sets the slider to a given tick. You can set these properties at both design time and at run time.

Percentile and TickIndex are updated as the user moves the slider. Percentile always indicates the slider value, in whole numbers, on a scale of 0 to 100 . Getting the TickIndex value returns the number of the closest tick to the current slider position. You can get the value of these properties using VRGet.

### Using a slider as a progress indicator

You can use a slider to indicate progress for some action. To do this, set the ReadOnly property to 0; this hides the slider arm. Then set RibbonStrip to 1; this causes the slider to fill in the slider shaft with the color specified by the RibbonColor property.

### Responding to the Track event

If your application needs to perform some action as the user drags the slider arm, then do it in the Track event for the slider.

## Value sets

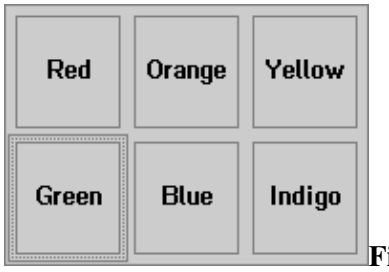

**Figure 59** A value set

ValueSet objects present you with a set of choices arranged in a grid, one of which is selected at all times. Value set items can be bitmaps, colors, icons, or text.

### Setting the number of rows and columns

Set the Rows and Columns properties to configure the number of rows and columns that the value set displays. You must set the rows and columns at design time; you cannot set them at run time.

### Setting item types

To set the types of all items at once, change the ItemType property. You can also set the types for a set of items using the SetAttributes method.

You can set the ItemType at design time or at run time.

### Setting item values

At design time, you can set the item values using the InitialList property . The syntax of each list item depends on the item type. See the ItemType section in the reference guide for a list of item types and allowable values.

At run time, you can set item values using the SetAttributes method.

### Handling the Click event

When a new value set item is selected, a Click event is generated. The index of the selected item is in the Selected property.

## Timers

You can use Timer objects to generate events as settable intervals. While enabled, a timer will continuously post Trigger events at the interval set in the Delay property.

### Setting the delay interval

To set the delay interval, set the Delay property to the number of milliseconds to wait before a Trigger event is posted. You can set this property at design time and at run time.

### Starting and stopping timers

A timer is started and stopped by setting its Enabled property. Initially timers are disabled, meaning that they will not generate Trigger events. To start a timer, set the Enabled property to 1. To stop a timer, set Enabled to

0 . If you set the Delay property while a timer is enabled, the previous delay is ignored and a Trigger event occurs once the new delay expires.

### Responding to

Trigger events

Trigger events occur after the delay interval has elapsed, measured from when the delay was set or when the timer was enabled. Because the OS/2 system may be engaged in some modal or time critical activities when the delay expires, your program may not process the Trigger event until some time after the delay has elapsed. As a result, your program should not count Trigger events as a way to keep track of time.

The following Trigger event routine changes the timer's Caption to the current time and date.

```
Trigger:
caption = Time('N') Date('N')
call VRSet 'TM_1', 'Caption', caption
return
```

## Dynamic Data Exchange (DDE) Client

DDE (Dynamic Data Exchange) is an OS/2 facility that allows applications to exchange data. Two applications engage in a DDE *conversation*, with one application being the DDE *client* (the one requesting the data), and the other a DDE *server* (the one responding to requests).

VX-REXX provides a DDE client, which allows your programs to talk to other applications, such as spreadsheets and word processors. It is possible, for example, to write a VX-REXX program that extracts data from a spreadsheet, then inserts that data into a word processing document.

For more information on the DDE client object, see Dynamic Data Exchange client.

## The VX-REXX console

VX-REXX includes a simple console window to which the output of **say** instructions and OS/2 commands is redirected. (I.e. output directed to the standard output devices STDOUT and STDERR is redirected to the VX-REXX console.) Although it handles most output the console is not intended to be equivalent to an OS/2 window or full-screen session. It does not handle special escape sequences of commands to position the cursor.

There is only one console window object per program, and it is referred to by its name, **Console**.

### Turning off the console

A VX-REXX program includes console support by default. The VRRedirectStdIO function is used to disable

console support:

```
call VRRedirectStdio 'Off'
```

The console cannot be referred to if it is disabled.

VRRedirectStdIO is also used to re-enable the console and/or to log its output to a file. If the console output is being generated by an OS/2 command, it may be more appropriate to prevent the command from writing to standard output rather than disabling the console window. For more information on redirecting the output from OS/2 commands see the chapter 'Controlling other programs'.

## Hiding, moving and clearing the console

If console support is enabled, the console window can be moved or sized as you would any other object:

```
call VRSet 'Console', 'Left', 2000, 'Top', 1000
call VRSet 'Console', 'Width', 1500, 'Height', 1000
```

To clear the console:

```
call VRMethod 'Console', 'Clear'
```

The console is temporarily hidden by setting its Visible property:

```
call VRSet 'Console', 'Visible', 0
```

Note, however, that the console will reappear as soon as a line of output is added.

## Removing the console from the OS/2 window list

The console may be removed from the OS/2 window list by setting its WindowListTitle property:

```
call VRSet 'Console', 'WindowListTitle', ''
```

## Console input

The console is also used for user input, but it must be made visible before the input is to occur. To do so, either set the Visible property to 1 or output a line of text using a **say** statement.

# Bitmaps, icons, and resources

This chapter explains how to add bitmaps and icons to your project.

## Loading bitmap and icon files

To display a bitmap file (.BMP) or icon file (.ICO) an object must support the PicturePath property. For bitmap files, PicturePath is the path of the bitmap to load, and must end in a .BMP extension. For example:

```
call VRSet VRWindow(), 'PicturePath', 'D:\OS2\BITMAP\OS2LOGO.BMP'
```

For icon files, PicturePath is the path of the icon to load, which must end in a .ICO extension. For example:

```
call VRSet 'IPB_1', 'PicturePath', 'D:\OS2\MDOS\WINOS2\WINOS2.ICO'
```

The icon *associated* with a non-icon file (executable files, for example) may also be loaded:

```
call VRSet 'IPB_1', 'PicturePath', 'D:\OS2\E.EXE'
```

The icon associated with a file is determined by OS/2. You can associate an icon with a file by defining a .ICON extended attribute for the file or by placing a .ICO file with the same filename in the file's directory. You can also associate an icon with a Workplace Shell file by using the **General** page of the file's **Settings** notebook.

The WindowIcon, Pointer, DefaultIcon and DragIcon properties are also used to load icons (and bitmaps in the case of DragIcon), using the same syntax as PicturePath.

## System icons and pointers

OS/2 defines a set of system icons that your project can use by simply setting PicturePath (or any of the other picture properties) to one of the following values:

Application The OS/2 application icon.

Information The OS/2 information icon.

Question The OS/2 question icon.

Error The OS/2 error icon.

Warning The OS/2 warning icon.

Illegal The OS/2 illegal operation icon.

File An icon representing a single file.

MultipleFiles An icon representing multiple files.

Folder An icon representing a folder.

Program A smaller icon for representing applications.

Arrow The default pointer icon.

Text The I-beam pointer icon (for text editing).

Wait The watch icon (for long operations).

Move A four-headed arrow.

SizeNWSE Two arrows, pointing northwest and southeast.

SizeNESW Two arrows, pointing northeast and southwest.

SizeWE Two arrows, pointing west and east.

SizeNS Two arrows, pointing north and south.

## Loading bitmap and icon resources

Bitmaps and icons can also be loaded from resources bound to an executable file (.EXE) or dynamic link library (.DLL). (A *resource* is binary data that has been appended to a file. A later section describes how to bind resources to your project). To load the resource you need to know what kind it is ( bitmap or icon), its resource number (an integer from 1 to 65000) and where it is located (in the executable or in a DLL).

For a bitmap resource, set PicturePath to the following format:

**#***resourceID***[:***DLLName***]**

For an icon resource, use the format:

**$***resourceID***[:***DLLName***]**

In both formats *resourceID* is the resource number and *DLLName* is the name of a DLL (do not include path information -- the DLL must be in the LIBPATH). If *DLLName* is omitted, then the resource is assumed to be in the project's executable (.EXE) file.

For example, to load bitmap with a resource number of 1899 from a DLL called **JHUTT.DLL:**

```
#1899:JHUTT
```

To load an icon with a resource number of 100 from the project executable :

```
$100
```

Resources can be used with any of the picture properties.

# Adding resources to your project

Resources can be added to a project by using the VX-REXX resource editor and the OS/2 resource compiler.

## Using the resource editor

The resources to be added to a project must be defined in a resource file ( .RC) in the project directory. This file is then processed by the OS/2 resource compiler to generate a binary resource file (.RES) that is later used by VX -REXX when creating an executable file from the project.

The VX-REXX resource editor is a simple text editor that allows you to edit a resource file. The resource file is a standard OS/2 resource file containing bitmap and icon definitions. A bitmap resource is defined as:

```
bitmap id filepath
```

For example:

```
bitmap 100 d:\os2\bitmap\os2logo.bmp
```

Icons are defined using a similar format:

```
icon id filepath
```

For example:

```
icon 342 d:\vxrexx\watcom2.ico
```

In either case, *id* is a number from 1 to 64000 that identifies the resource and *filepath* is the path of the bitmap or icon file.

To run the VX-REXX resource editor, select the **Open** menu after clicking mouse button 2 on a window, or select the **Resource editor** item in the **Project** menu.

The VX-REXX resource editor creates a file in the project's source directory, so it will not work on a project that was not created from a template and has never been saved. You must save such projects before running the resource editor .

If you do not wish to edit the resource file directly, click mouse button 2 on the resource editor to invoke its pop-up menu and choose **Insert resource** to display a dialog that prompts you for the necessary information. You may also drag bitmap and icon files from the Workplace Shell and drop them on the resource editor .

After making modifications, save the resource file to disk and compile it with the resource compiler by selecting **Save and compile** from the **File** menu. If errors occur, correct them and repeat the process.

## Resource binding

Once a binary resource file (.RES) has been created in your project directory, VX-REXX will bind that file to your project when an executable is made. The resources will then be available to the executable when it is run, using the syntax described in the previous section.

Resources are also available when testing a project using **Run project**.

In both cases, VX-REXX displays a message box describing the resource binding process while it occurs.

Resources are *not* available to your project when **Debug project** is used.

## Editing bitmaps and icons

OS/2 includes an icon editor in the **Productivity** folder in the **OS/2 System** folder. (Or type **iconedit** at the OS/2 command prompt). This is the same editor that is used to edit Workplace Shell icons.

Small bitmaps can also be edited using the icon editor. For larger bitmaps, you must use another application. One possibility is the Microsoft Windows **Paint Brush** application -- VX-REXX can load Windows bitmaps as well as OS/2 bitmaps .

## A note about icon formats

An icon file or resource can contain icons for various screen resolutions and color palettes. At least one of these formats will be the 'Independent Color Form' that will be used when there is no exact match for the screen resolution . However OS/2 will not display your bitmap in all cases unless you include at least two formats. The '8514 - 16 colors' is a good choice for the second format.

## The executable icon

If your project's resources include at least one icon, the Workplace Shell will use the icon with the lowest resource number as the default icon when its executable is displayed in a folder or on the desktop.

# Drag and drop operations

This chapter explains how to add drag and drop support to your project. Drag and drop operations are available on most objects.

## Drag and drop definitions

*Drag and drop* is a form of direct manipulation where the user selects an object, drags it across the screen and drops it on another object. The drag operation is usually started by pressing mouse button 2 over an object, moving the mouse to drag the object, and then releasing mouse button 2 to drop the object. The Workplace Shell uses drag and drop operations extensively, and these capabilities are easily added to your own project.

Before adding drag and drop support, it is important that you understand the terminology that will be used throughout this chapter.

### Basic terminology

A *drag item* is something being dragged. There is at least one drag item in every drag and drop operation, though there may be more than one. Each drag item represents a thing to be dragged -- an object in your program, or a record in a container, or a Workplace Shell object, or a file, or something else.

A *drag source* is something that allows a drag operation to start. The user presses mouse button 2 on the drag source to start a drag and drop operation . Most VX-REXX objects can act as drag sources through the use of the AllowDrag property.

A *drag target* is something that allows a drag item or items to be dropped on it. Not all drag items can be dropped on all drag targets -- the drag target chooses which are allowable. Most VX-REXX objects can act as drag targets through the use of the DragTarget property.

The *operation type* defines the drag and drop operation that is taking place . Drag items can be *moved*, *copied* or *linked*. The definition of each operation type depends on the drag source, the drag target and the format of the drag items . If the drag items, drag source and drag target are all Workplace Shell objects, a move operation relocates an object, a copy operation replicates the object, and a link operation shadows the object. Users can specify a particular operation by holding down the **Ctrl** and/or **Shift** keys. If no key is pressed, a default operation is chosen by the drag target. The visual appearance of drag items changes depending on the operation.

## Adding drag targets to your project

Most VX-REXX objects are drag targets. The DragTarget property describes the kinds of drag items that can be dropped on an object. If a drag item is dropped, the drag target receives a DragDrop event.

The user cannot drop drag items onto an object if those items do not match the criteria defined by the DragTarget property. Items can only be dropped if they match the DragTarget settings of the drag target *and* the drag target has a DragDrop event defined.

An object can also explicitly refuse any drops by setting its DragTarget property to **None**.

## Files, records and objects

The three most common settings for DragTarget are:

Files='*' Accepts all files, either from Workplace Shell objects corresponding to a file, or container records that have the Filename attribute set. One or more filemasks (using the usual OS/2 wildcard characters ***** and **?**) can be specified, and only files which match those masks will be allowed to be dropped on the object.

For example, use **Files='*'** to allow all files. Use **Files='* .C','*.H'** to allow all files ending in **.C** and **.H** extensions.

Records Accepts records dragged from a container in the same project.

Objects Accepts drag items generated by other objects in the same project .

You can combine any or all three using semicolons. For example, **Files= '*'semi.Records** accepts all files and all container records. The setting **All** is equivalent to **Files='*';Records;Objects** and is the default value for DragTarget.

If the **Files** setting is specified, the **Types** setting may also be specified to allow only files with certain OS/2 file types to be dropped. For example, the setting **Files='*';Types='Plain text','OS/2 Command File'** only allows text and command files to be dropped.

## Default and supported operations

The default operation in a drop is determined solely by the drag target. If not specified, **Move** is assumed to be the default operation. A different default operation (**Copy** or **Link**) may be specified in the DragTarget using the form **Default=Link** or **Default=Copy**. For example:

```
Files='*.TXT';Default=Copy
```

```
The drag target may also limit the allowable operations using Operations
:
```

```
Records;Default=Copy;Operations=Move,Copy
```

```
All operations are supported by default.
```

## Other formats

If none of the formats listed above are sufficient, you may specify the exact OS/2 rendering formats the target will accept using the **Formats** keyword. When specifying a rendering format, you may not specify **Files**,

**Records** or **Objects**. If you wish to handle objects of these types, you must ensure that your **Formats** string specifies the appropriate underlying rendering formats.

The rendering format describes how the drag source and the drag target are to exchange the data that the drag item represents. The rendering format is actually a string of pairs of data. Each pair describes a *mechanism* for exchanging data and a *format* to use with that mechanism. The **DRM_OS2FILE** is the most common mechanism, representing an OS/2 file. VX-REXX also defines a **DRM_VXREXX_OBJECT** mechanism. Drag items with the **DRM_PRINT** and **DRM_DISCARD** mechanisms may be dropped on Workplace Shell printer and shredder objects, respectively.

The syntax for rendering mechanisms and formats is either a list of pairs :

```
<mech1,format1>,<mech2,format2>,...
```

or a *cross-product*:

```
(mech1,mech2,...)x(format1,format2,...)
```

For example, a Workplace Shell folder is represented as:

```
<DRM_OS2FILE,DRF_UNKNOWN>,<DRM_OBJECT,DRF_OBJECT>
```

To accept any and all Workplace Shell objects, even those that do not represent files, you may specify the string:

```
Formats=(DRM_OS2FILE,DRM_OBJECT)x(DRF_UNKNOWN)
```

The default rendering format for VX-REXX objects other than containers is :

```
(DRM_VXREXX_OBJECT,DRM_PRINT,DRM_DISCARD)x(DRF_UNKNOWN)
```

More detailed information on types, mechanisms and formats can be found in the programming guides for OS/2.

## Passing the drop to the parent object

If there are several objects that you wish to designate as drag targets, it may be more convenient to set their DragTarget properties to **Parent** instead. The parent object will then control and receive the result of any drag and drop operations. You must also define the DragTarget property and DragDrop event on the parent object. The internal name of the actual target object will be available in the DragDrop event.

### Container restrictions

Each record in a Container object is a drag target. You may further restrict the types of drag items that may be dropped on a record by setting its Target attribute using SetRecordAttr. The syntax for Target is similar to that of DragTarget except that the **Formats**, **Parent**, **Default** and **Operations** values may not be specified . If the Target attribute of a record is not set, it defaults to **All**.

## Handling drops

Each drag item that is dropped on an object is really a *request* to perform an action. It is the responsibility of your program to recognize, interpret and act on the request. There is one exception to this rule: when a record is dragged to a new position in the same container, the record is automatically repositioned unless a MoveRecord event has been defined.

### Programming with the DragDrop event

When a DragDrop event occurs, use VRInfo to retrieve information about the drag item that was dropped. See the online **VX-REXX Reference** section for DragDrop for a complete list of available information. Here are some tips:

o**SourceObject** identifies the object in your program that was the drag source . If it is null, then the drag source was not part of your project.

o**Object** identifies the object processing the DragDrop event. **TargetObject** identifies the object underneath the drop point. They are usually the same unless the **Parent** setting for DragTarget is used.

oIf the drag source is a container in your project, then **SourceRecord** identifies the record that is being dragged.

oIf the drag source is a ListBox, ComboBox or ValueSet in your project, then **SourceIndex** identifies the item being dragged.

oIf the drag target is a Container, then **TargetRecord** identifies the record underneath the drop point.

oIf the drag target is a ListBox, ComboBox or ValueSet, then **TargetIndex** identifies the item underneath the drop point.

By carefully choosing the values for DragTarget, programming the DragDrop event is quite simple. For example, you can add the ability to drop files onto a multiline entry field by setting DragTarget to **Files** and using the following DragDrop event routine:

```
MLE_1_DragDrop:
file = VRInfo( 'SourceFile' )
if( VRFileExists( file ) ) then do
contents = CharIn( file, 1, Chars( file ) )
call Stream file, 'C', 'Close'
call VRMethod 'MLE_1', 'Insert', contents
```

```
end
return
```

The Insert method is used to insert the text of the file into the MLE.

See the section on dragging and dropping records in the 'Using objects ' chapter for information on using the DragDrop event with Container objects.

### Programming with the MoveRecord event

The MoveRecord event occurs when the user drags a record to a new position within the same container. For more information on the MoveRecord, see the ' Using objects' chapter.

## Adding drag sources to your project

Most VX-REXX objects can also act as drag sources by setting the AllowDrag property to 1. Then when the user clicks mouse button 2 on the object and moves the mouse, a DragStart event occurs. If you have defined a DragStart event, the drag operation will start when you call the StartDrag method. If no DragStart event is defined, the drag operation will start immediately.

If a Container object is to be a drag source, please refer to the 'Using objects' chapter. The remainder of this section applies only to objects other than the container.

### The StartDrag method

For objects other than containers, the StartDrag method can be used to fully describe a drag item and to start the drag operation. Although the StartDrag method has many arguments which allow you to specify details such as the default operation , drag icon, and rendering format, you will usually be able to use the default values.

The StartDrag method returns a value which indicates whether or not the drag item was dropped on a drag target, and if so identifies the target.

For full information on the StartDrag method, see the online *VX¨REXX Reference*.

### Dragging objects

Objects in your project can be dragged onto other objects in your project with very little work. The objects that are to act as drag targets should include **Objects** in their DragTarget property and define an appropriate DragDrop event . The objects that are to be drag sources need only set AllowDrag to 1.

If you specify the rendering format when calling the StartDrag method, you must include the pair:

```
<DRM_VXREXX_OBJECT,DRF_UNKNOWN>
```

If this format is not specified, the drag targets will not recognize the drag item as a VX-REXX object from the same project.

## Dragging files

A drag item that represents a file can be dragged onto drag targets in other applications and on the Workplace Shell. The StartDrag method must be invoked with the following values:

oThe *container name* argument (not to be confused with a container object) must be set to the drive and directory containing the file, including the trailing backslash. For example, 'D:\OS2\BITMAP\'.

oThe *source name* must be set to the *physical* name of the file. (The physical name of a file may be different than its folder name on systems that do not support long filenames). For example, 'OS2LOGO.BMP'.

oThe *target name* must be set to the suggested name of the file. (On systems that do not support long filenames at the file level, this should be the long version of the filename. On other systems it is usually the same as the source name ). For example, 'OS2LOGO.BMP'.

oThe *rendering format* must include the **DRM_OS2FILE** mechanism. For example , **<DRM_OS2FILE,DRF_TEXT>**. Including **DRM_PRINT** and **DRM_DISCARD** allows the file to be dropped on printer and shredder objects.

oThe *type* describes the type of the file. (If type is unknown or not important, use 'Unknown'). For example, 'Plain Text'.

Here is an example:

```
PB_1_DragStart:
call VRMethod 'PB_1', 'StartDrag', 'Move', 'File', ,
'<DRM_OS2FILE,DRF_BITMAP>', 'Bitmap', ,
'D:\OS2\BITMAP\', 'OS2LOGO.BMP', 'OS2LOGO.BMP'
return
```

What happens to the file after it has been dropped depends on the drag target . The return value from the StartDrag method lets you know if the drag item was dropped or the drag operation was cancelled. If a drop occurred, it is the drag target's responsibility to handle the file.

## Dragging to the shredder and printer

Drag items may be dropped on the Workplace Shell shredder object if the **DRM_ DISCARD** rendering format is used, and on Workplace Shell printer objects if the **DRM_ PRINT** rendering format is used. Unlike

other drag targets, dropping a drag item on these objects does not generate a DragDrop event. A DragDiscard event occurs if an item is dropped on the shredder and a DragPrint event occurs if an item is dropped on a printer object.

Dropping an item on the shredder or a printer object only results in a notification being sent to your project. You are responsible for doing any actual discarding or printing using the information provided in the DragDiscard and DragPrint events.

# Adding menus to a program

Many applications have menus from which you can choose options or actions . For example, you have used the **Tools** menu in VX-REXX to choose objects and the **Project** menu to save a project. This chapter shows you how to add a menu to a window using the VX-REXX menu editor.

## Types of menus

### Menu bar and pull-down menus

Typical application windows have a *menu bar* which contains one or more *pull- down menus*. The menu bar is displayed immediately below the window title bar . For example, Figure 60 shows the VX-REXX menu bar.

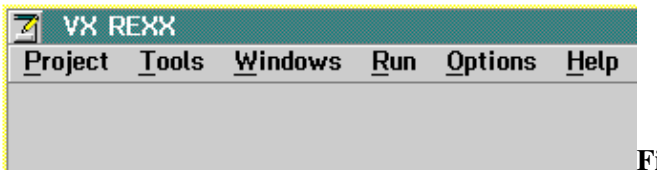


**Figure 60** VX-REXX menu bar

Pull-down menus such as **Help** are common to many applications. Other pull- down menus such as **Tools** and **Run** in VX-REXX are application specific.

To open a pull-down menu, click on the menu name. This drops down a list of *menu items*. For example, if you click on the VX-REXX **Project** menu, the menu shown in Figure 61 appears.

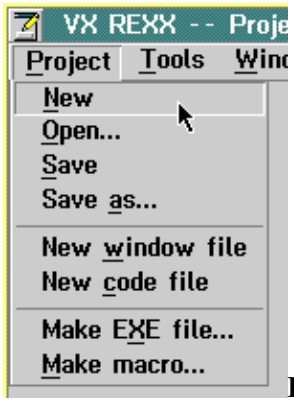


**Figure 61** VX-REXX Project menu

An underlined letter in a menu name or menu item is called a *mnemonic*. It is a the key that can be pressed to choose the menu or menu item instead of using the mouse. To choose a menu using a mnemonic you must first activate the menu bar by pressing the **Alt** key. To use a mnemonic to choose a menu item you must first open the menu that contains the item.

A key called an *accelerator* may be shown to the right of a menu item. An accelerator is similar to a mnemonic in that it allows the user to press a key instead of clicking on the menu item. It is different in that the menu does not have to be open for the accelerator to work. This can save a lot of mouse clicking for users familiar with an application.

The horizontal lines are separator bars, and are used to group choices to make menus easier to read. Choosing a separator does not result in any action .

Some items in a menu perform an action immediately when chosen; for example, **Save** saves the current project. By convention, a menu item which contains an ellipsis (...), such as **Save as...**, indicates that a dialog will appear when the item is chosen, instead of an immediate action.

## Pop-up or context menus

Pop-up menus (also called context menus) are similar to pull-down menus except that they are not attached to a menu bar. Pop-up menus can be positioned anywhere on a window. Usually pop-up menus are associated with a specific object such as a record in a container, and are displayed when you click on the object with mouse button 2 or when you press **Shift+F10** when the object has the focus.

Figure 62 shows a pop-up menu.

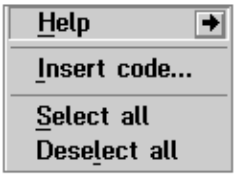


**Figure 62** A pop-up menu

## Cascaded menus

A menu item that contains an arrow pointing to the right indicates that a *cascaded menu* will drop down when the item is chosen. The cascaded menu is displayed beside the chosen item as in the sample menu shown in Figure 63.

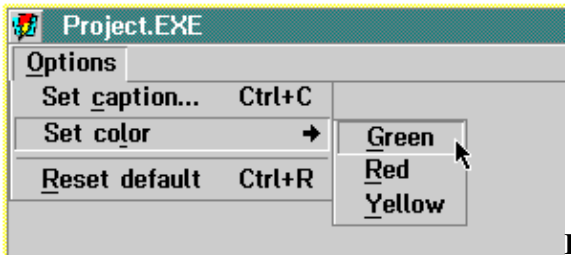


**Figure 63** Cascaded menu

A cascaded menu is used to reduce the length of a menu by grouping related choices. Although you can display up to 5 levels of cascaded menus, using more than 3 levels will make the menus difficult to use.

If the arrow in a menu item is two dimensional, then the menu is an unconditional cascaded menu. This means that the menu automatically becomes visible when the item is selected. If the arrow is three dimensional, then the menu is a *conditional cascaded menu*. For these menu items, the cascaded menu does not appear automatically. The three dimensional button indicates that one of the cascaded menu items is preselected. Choosing the menu item with the arrow is the same as choosing the preselected item. The user can display a conditional cascaded menu by clicking on the arrow button.

Figure 64 shows a conditional cascaded menu after the arrow button has been clicked. The menu item **General help** has been preselected, as indicated by the check mark.

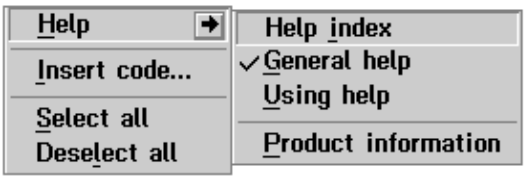


**Figure 64** Conditional cascaded menu

## Menu properties

In VX-REXX, a menu is made of objects. The menu bar itself is a MenuBar object. It contains Menu objects which in turn contain MenuItem objects and possibly other Menu objects (in cascaded menus). At design time, menus are created and modified using a menu editor, which is described later in this chapter. At run time the properties of the menu objects, described below, can be changed using the VRSet and VRGet functions. This is also described later in this chapter .

Accelerator The Accelerator property is a string value which represents the keyboard equivalent for a given menu item. VX-REXX has many strings to represent special keys, such as **{Alt}** for **Alt**, and **{Ctrl}** for **Ctrl**. A complete list of these key strings is available in the description of the KeyString property in the online *VX¨REXX Reference*. To change accelerators at run time, change the Accelerator property and then invoke the InstallAccelerators method.

Caption The Caption property is the text that appears on the menu bar or within a menu. The position of items on a menu bar and the width of menus is automatically adjusted to fit the captions. The order of items in a menu bar or within a menu is controlled by the order in which the items are created. This is managed for you by the menu editor.

To define a character in a caption as a mnemonic, precede the character with a tilde (~). To create a menu separator, set the caption to a single hyphen ( -).

Checked The Checked property is used to indicate that an option is in effect . A menu item that is checked has a check mark displayed beside it when the menu is dropped down. The values of both the enabled and checked properties can be queried and changed at run time.

DefaultItem The DefaultItem is used to indicate the default selection for a conditional cascaded menu. The default item is automatically checked.

Enabled The Enabled property controls whether the menu item can be chosen when the application is run. Menu items are enabled by default. You may want to disable a menu item to prevent an action until another

action is performed. A disabled menu item is dimmed as a visual cue that its action cannot be chosen .

HelpTag, HelpText, HintText You can provide help for your menu items as you can for most objects. The HelpTag, HelpText, and HintText properties are all editable from within the menu editor.

Name The Name property can be used to refer to a menu or menu item from a routine. A default name will be generated when you create each item. You can change the default name to make your program more readable.

Visible Menu objects can be made invisible by setting their Visible property to 0. You may want to do this to hide a menu which is only used as a pop-up menu.

## Menu editor

You create menus in VX-REXX using the menu editor. To open the editor, display a window's pop-up menu, select **Open...** and choose **Menu editor** . The menu editor is shown in Figure 65.

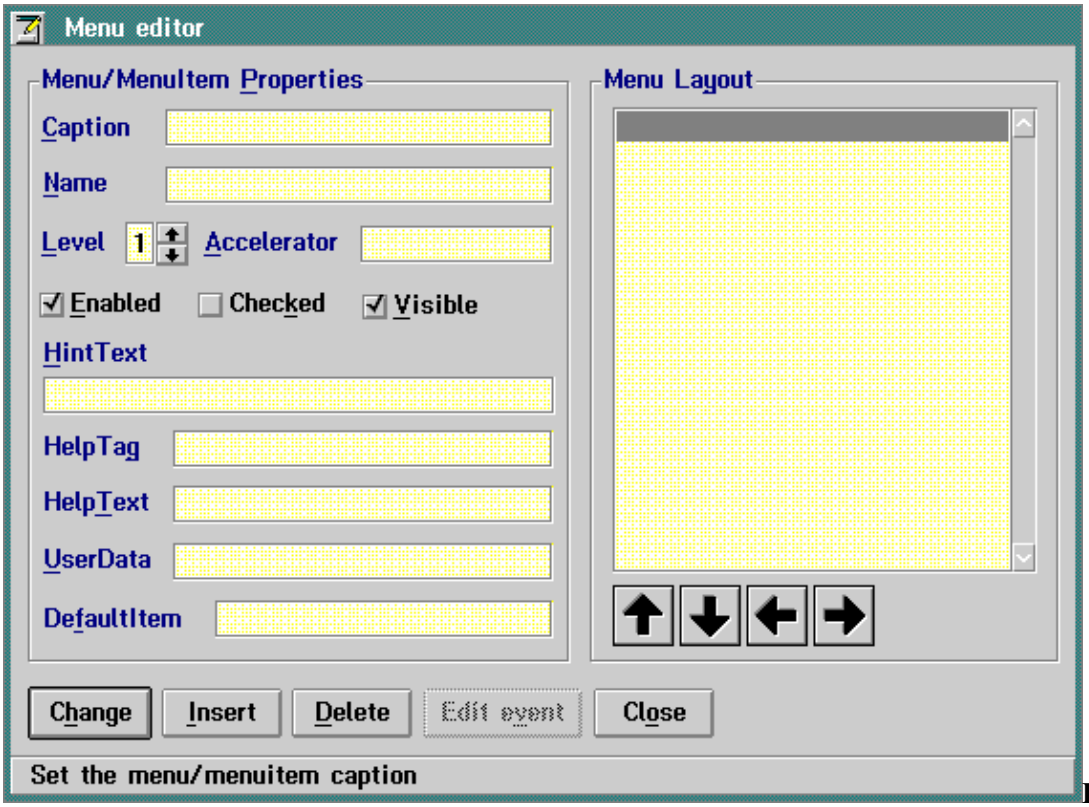


**Figure 65** Menu editor

The entry fields and check boxes are used to create the menus and set the properties. The **Change**, **Insert** and **Delete** push buttons modify, add or remove items from the menu structure displayed in the list box. The four arrow push buttons let you move the order and level of menu items displayed in the Menu Layout list box. The **Edit event** push button opens the edit window so that you can add code to a menu item. The **Close** button saves the changes to your project window and closes the menu editor.

Pull-down menus, pop-up menus, menu items, and cascaded menus are all created using the menu editor. You create each one by entering a caption and choosing a level. For example, the caption of an item created at level 1 is the name of a drop-down menu that will appear in the menu bar. The items in the drop down menu will be the items added at level 2 underneath the level 1 item.

The following sections show, by example, how to use the menu editor.

## Creating a menu

This section describes how to create the sample drop-down menu **Options** shown in Figure 66.

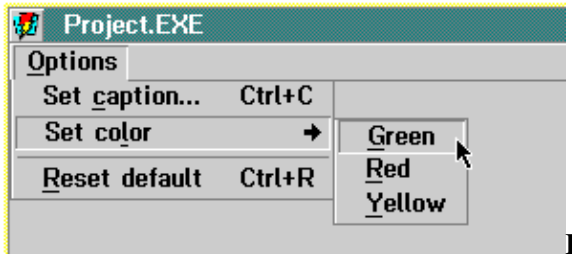


**Figure 66** Options menu

The sample menu contains commands to change the caption and color of a push button. The menu has three items: **Set caption...**, **Set color** and **Reset default**. The ellipsis (...) following **Set caption** indicates that a dialog is displayed when you choose this item. The arrow beside **Set color** indicates that a cascaded menu appears when this item is chosen. When the user chooses **Reset default** an action is immediately run; in this case the push button's caption and color are reset to the default values.

To create this menu:

1.Display the window's pop-up menu.

2.Choose **Menu editor**.

3.Type **~Options** in the **Caption** entry field. The tilde character (~) signifies that when the menu is displayed, the character which follows will be underlined . Pressing that key is the keyboard equivalent to clicking on the item to choose it.

4.Move to the **Name** entry field and type **MenuOptions**. You can enter any value for the name, but choosing a name related to its use makes it easier to remember and makes the code more readable. If you do not enter a name, a default name will be generated when it is inserted in the menu structure.

5.Press **Enter** or click on **Change** to add it to the list box with default values for all other properties. When it is added to the list box, the caption and name fields are cleared and a new blank entry is added to the end of the list in the list box.

By default the menu is added at level 1 which means its caption will appear on the window's menu bar. Also, it is enabled and is not checked by default.

The next steps add the items to the drop-down menu. They will be entered at level 2. The three items are **Set caption**, **Set color** and **Reset default**. A separator line is to be included between **Set color** and **Reset default**.

1.Type **Set ~caption...** in the **Caption** entry field.

2.Move to the **Name** entry field and type **MenuOptionsCaption**.

3.Move to the **Level** entry field and replace the current value with 2.

4.Move to the **Accelerator** field, and type **{Ctrl}C**. This accelerator string means that the event routine associated with clicking on the **Set caption** menu item can be executed even when the **Options** menu is not opened by pressing the **Ctrl** key, and the letter **C**.

5.Press **Enter** or click on **Change**. The menu item is added to the list box. The item will be indented below the first item to visually denote its level.

6.Type **Set co~lor** in the **Caption** entry field. Because the key 'c ' is the mnemonic key for **Set caption**, a different key must be selected for **Set color**. In this case the letter 'l'.

7.Press **Enter** or click on **Change**. No Name property was entered so a default name will be generated. The generated name will be 'Menu' concatenated with a number to create a unique name (e.g. Menu3). The default name is satisfactory because no click event will be associated with this menu item . When the item is clicked a cascaded menu appears.

8.Type **-** (hyphen) in the **Caption** entry field. A hyphen adds a separator between menu items. No Name property is set because the item cannot be manipulated.

9.Press **Enter** or click on **Change**.

10.Repeat steps 1 through 4 setting the caption to **~Reset default**, the name to **MenuOptionsReset**, and the accelerator to **{Ctrl}R**.

The menu editor window should look like Figure 67.

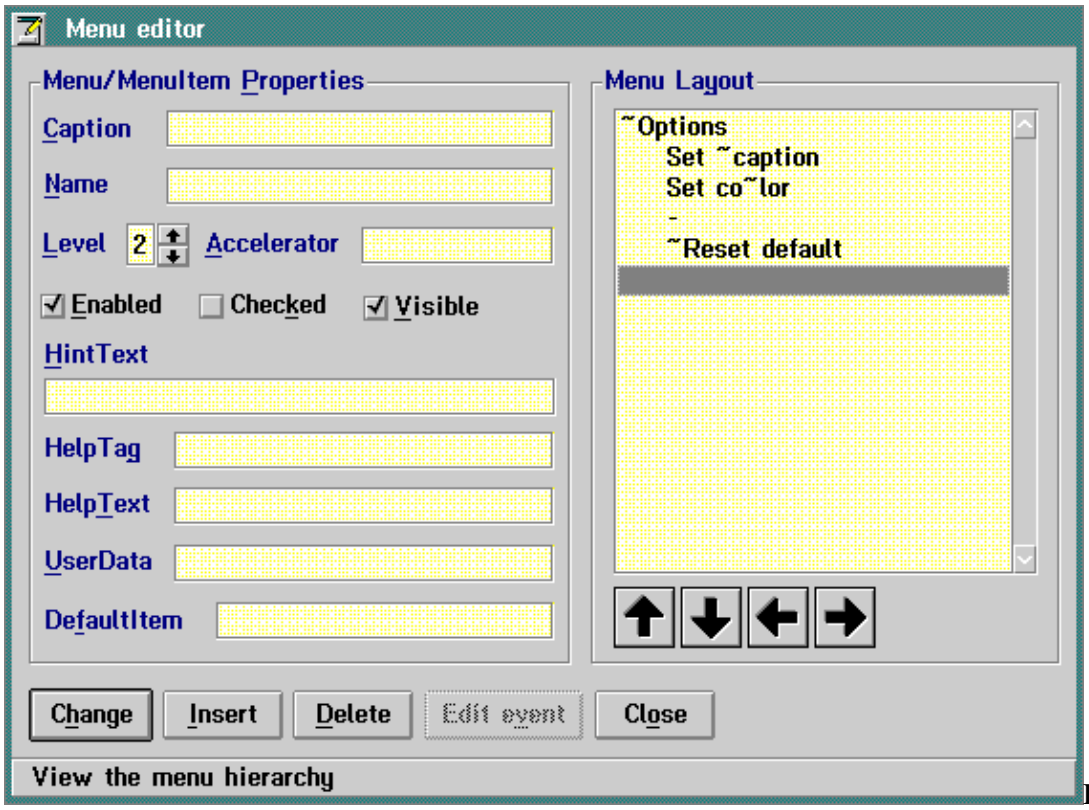


**Figure 67** Level 1 and
2 menu items

To complete the menu you need to add a cascaded menu for the menu item **Set color**. It contains a list of colors from which you can choose one. The items in the menu will be at level 3 and are inserted between **Set color** and the separator. Items are inserted before the item currently selected in the list box.

To add the **Set color** menu:

1.Click on the separator line (hyphen) in the list box. The **Caption** entry field will contain a '-'. You will add the level 3 items above this item.

2.Replace the caption with **~Green**.

3.Move to the **Name** entry field and replace the name with **MenuOptionsGreen** .

4.Move to the **Level** entry field and replace the value with **3**.

5.Click **Insert**. A new entry will be added to the list box. This entry will be identical to the entry that was selected when the **Insert** button was pressed.

6.Repeat steps 2 through 5 for the colors **Red** and **Yellow** adding a **~** at the beginning of each color name and adding the appropriate name entry.

When all of the colors have been added the menu structure is complete. Because **Set color** is followed by level 3 menu items, an arrow will automatically be displayed beside its caption when the application is run.

As yet, none of the menu items perform any action. The next section shows you how to add code to the menu items.

## Adding code to menu items

Code is attached to objects to handle events. The Click event is the only event available for a menu item.

You add code to menu items in the same way as you add code to objects except that you open the section editor from the menu editor. Instead of selecting an object in a window, you select a menu item from the list box in the menu editor. To edit code for that item's Click event, press the **Edit event** button, or double click on the menu item. The editor will contain the event routine for the selected item. You can then edit the event routine.

For example, to add code for the **Reset default** item:

1.Select **Reset default** from the list.

2.Click on **Edit event**. The section editor is opened as in Figure 68 .

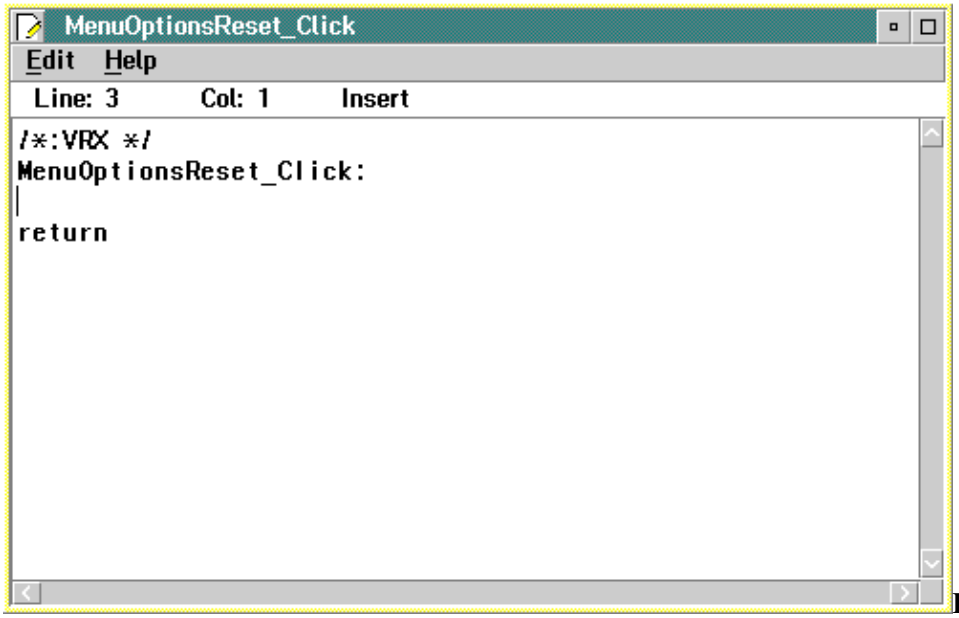


**Figure 68** Edit a menu event

The section name is generated from the menu item name, MenuOptionsReset, and the event name, Click.

You add the code for the event between the section name and the return statement. For example, the code for this menu item sets the caption on the push button with the name PB_PushMe to the default value **Push Me!** and the color of the push button to the system default color. The code added is:

```
call VRSet 'PB_PushMe', 'BackColor', '<default>'
call VRSet 'PB_PushMe', 'Caption', 'Push Me!'
```

After you add the code, close the section editor and return to the menu editor.

The code for each of the items in the cascaded menu is similar since each one performs the same action; that is, it sets the color of the push button. The code for MenuOptionsGreen is:

```
call VRSet 'PB_PushMe', 'BackColor', 'Green'
```

The VRSet procedure sets the push button's color. The code for the click events for each of the other colors would be the same, replacing **Green** with **Red** and **Yellow**.

When you are finished creating your menu, click on **Close** to apply the changes and close the menu editor window.

## Creating a pop-up menu

Any menu can be displayed as a pop-up menu. To do this, invoke the Popup method on the menu. Usually, your program will do this in response to a ContextMenu event which is generated when the user clicks mouse button 2 on an object.

The user can either choose one of the menu items or dismiss the menu by clicking outside the menu or pressing **Esc**. If the user clicks on a menu item, a Click event is generated for that menu item in the same way as if the menu was attached to a menu bar.

The following ContextMenu event routine displays the menu created in the previous section as a pop-up menu.

```
PB1 ContextMenu:
call VRMethod 'MenuOptions', 'Popup'
return
```

All menus that you use as pop-up menus must be associated with some window 's menu bar. If you do not want these menus to appear on the menu bar, uncheck the Visible box for that menu in the menu editor. This will remove the menu from the menu bar, although you can still use it as a pop-up menu. If none of the menus attached to a menu bar are visible, the menu bar itself will be invisible .

## Responding to a pop-up menu

In the Click event for a menu item, the internal name of the object for which the ContextMenu event was generated is available. To get it, pass the keyword **Source** to VRInfo. The action on the menu item should be applied to the source object. For example, the **Popup** sample program changes the BackColor property of a button with the following code:

```
MI_BackGreenClick: /* menu item click routine */
call SetColor 'Green'
return

SetColor: procedure
parse arg color

button = VRInfo( 'Source' )
call VRSet button, 'BackColor', color
return
```

In the **SetColor** routine, the call to VRInfo gets the name of the button on which the user clicked mouse button 2.

## Using pop-up menus with containers

If the user caused a ContextMenu event for a container, then the menu item Click routine is more involved. The container automatically marks the records to which the pop-up menu should apply. The Click event can get the list of records by using the GetRecordList container method to list all records that have the *source emphasis* (ie. the Source attribute set 1.) The Click event code must reset the Source attribute back to 0.

If no records have the source emphasis, the Click routine must check if the container itself has the source emphasis. It can do this by getting the container 's Source property. If the container itself has the source emphasis, then the menu action does not apply to any particular record, but rather to the container as a whole. Again, the Click routine must turn off the Source property if it is on.

For example, suppose you have a pop-up menu that has two items: **Open** and **Help**. The program uses the Popup method to display the menu in response to a ContextMenu event on a container. If the user picks the **Open** menu item and the user clicked on a record, then information about that record is displayed. If the user picked **Help**, then help for that record is displayed. If the user clicks mouse button 2 on the container white space, then picking **Open** does nothing, but picking **Help** brings up some general help.

Here are the two menu item click routines that would implement this behavior :

```
MI_Open_Click:
ctn = VRInfo( 'Source' )

/* Get the list of records with source emphasis
*/
call VRMethod ctn, 'GetRecordList', 'Source', 'reclist.'
do i = 1 to reclist.0
```

```
/*
... display information about the record ...
*/

/* Reset the record's source attribute
*/
call VRMethod ctn, 'SetRecordAttr', reclist.i, 'Source', 0
end

/* Turn off source emphasis for the container in case
it is currently on.
*/
call VRSet ctn, 'Source', 0

return



MI_Help_Click:
ctn = VRInfo( 'Source' )

/* Get the list of records with source emphasis
*/
call VRMethod ctn, 'GetRecordList', 'Source', 'reclist.'
do i = 1 to reclist.0

/*
... display help for the record ...
*/

/* Reset the record's source attribute
*/
call VRMethod ctn, 'SetRecordAttr', reclist.i, 'Source', 0
end

/* Turn off source emphasis for the container in case
it is currently on.
*/
if VRGet( ctn, 'Source' ) = 1 then do

/* ... display help for the record ...
*/

/* Reset the container's Source property
*/
call VRSet ctn, 'Source', 0
end

return
```

For details on the way containers set the Source attribute see

You can use a menu both as a pop-up and a pull-down menu if you make the following change to your code: instead of getting the list of records that have the source emphasis, get the list of records that have either source or the selected emphasis. From the example above, you would change:

```
call VRMethod ctn, 'GetRecordList', 'Source', 'reclist.'
```

to

```
call VRMethod ctn, 'GetRecordList', 'SourceOrSelected', 'reclist.'
```

Getting the **SourceOrSelected** list returns the records that have the source emphasis, or if there are none, then the records that are selected. Remember that pull-down menus should apply only to the selected records.

## Creating a conditional cascaded menu

To make a cascaded menu into a conditional cascaded menu, set the DefaultItem property for that menu to the name of the default menu item. You do not have to check any of the menu items; the menu will automatically check the default item. A Click event for the default item is generated if the user clicks the menu item without displaying the cascaded portion.

You can set the DefaultItem for a menu using the menu editor.

## Changing menus at run time

The Caption, Checked, Enabled, and Visible properties can be changed at run time. This enables you to change your menus to reflect a change in the state of the program. For example, you can disable a menu item when it is not possible to perform that action for some reason.

A convenient place to put the code to enable and disable menu items is in the Click event for the menu containing the items. This event will occur whenever the menu is displayed.

### Checking menu items

Menu item properties are accessed with VRGet and VRSet just like any other object's properties. For example, to set a check mark beside the **Green** option in our sample menu you would use the following code:

```
call VRSet 'MenuOptionsGreen', 'Checked', 1
```

```
At the same time you would uncheck the other items:

call VRSet 'MenuOptionsRed', 'Checked', 0
call VRSet 'MenuOptionsYellow', 'Checked', 0
```

## Installing accelerators at run time

The Accelerator properties of the menu items are ineffective until the window builds an accelerator table. This is an internal data structure which the window uses to control the use of accelerator keys. To create this table, you must use the window object's InstallAccelerators method. All applications created in the VX-REXX environment automatically call this method at run time. You only have to call this method if you wish to change the accelerators for a menu item , or add new menu items with accelerators.

# Using built-in dialogs and system functions

In addition to providing functions like VRGet and VRSet for manipulating objects, VX-REXX also defines functions for displaying dialogs and for file system manipulation.

## Predefined dialogs

Four functions are available for displaying predefined dialogs:

oVRFileDialog displays an OS/2 file dialog, to prompt the user for one or more file names

oVRFontDialog displays an OS/2 font dialog, to prompt the user for a font name and style

oVRMessage displays a message and waits for user confirmation

oVRPrompt prompts the user for a string

These functions do not return until the dialog is dismissed. The first parameter is the name of the window that is to own the dialog -- the owner and any of its children will be disabled while the dialog is active.

Each of these functions along with their parameters is described in the online *VX¨REXX Reference*.. Predefined dialogs can save you programming time. They also help you maintain a consistent look in your applications. The following sections describe these dialogs and show how you may use them.

### File selection dialog

The file selection dialog (VRFileDialog) is used to obtain a file specification for file operations such as reading or writing. Figure 69 shows a sample file dialog.

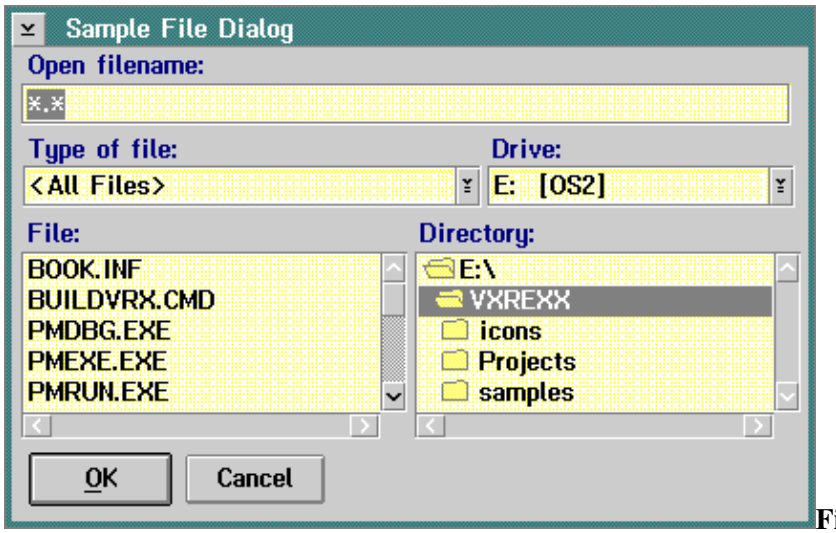


**Figure 69** File dialog with default values

The dialog displays several fields related to the file: its name, type, and location. The REXX code to display the dialog is:

```
path = VRFileDialog( VRWindow(), 'Sample File Dialog' )
```

VRFileDialog provides default values for all of the fields. By default, all files in the project's working directory are displayed in the file list box. The only parameters that must be supplied are the reference to the window to which the dialog is attached and a title for the dialog. In this example, the reference to the window is obtained using the function VRWindow and the dialog title is **Sample File Dialog**. The full path and name of the chosen file is returned.

If the default values are not appropriate for your application, you can include parameters to override them. For example, you may want the dialog to display only files with a particular extension. The following sample code for an open file dialog includes the dialog type and initial path parameters so that only files in the project working directory with an extension **BMP** are listed. Note that the comma at the end of the first line is the REXX line continuation character .

```
path = VRFileDialog( VRWindow(), 'Select a Bitmap file', ,
'Open', '*.BMP' )
```

The dialog type parameter sets the string that is displayed above the file name entry field. In this example the string is 'Open filename:' . The initial path determines the mask for the file name and the values displayed for the drive, directory and file boxes. If no path is included, the application's working directory is assumed. If the working directory is not suitable, you can include a complete path name for this parameter. For example,

```
path = VRFileDialog( VRWindow(), 'Select a Bitmap file', ,
'Open', 'E:\OS2\BITMAP\*.BMP' )
```

would produce the dialog shown in Figure 70.

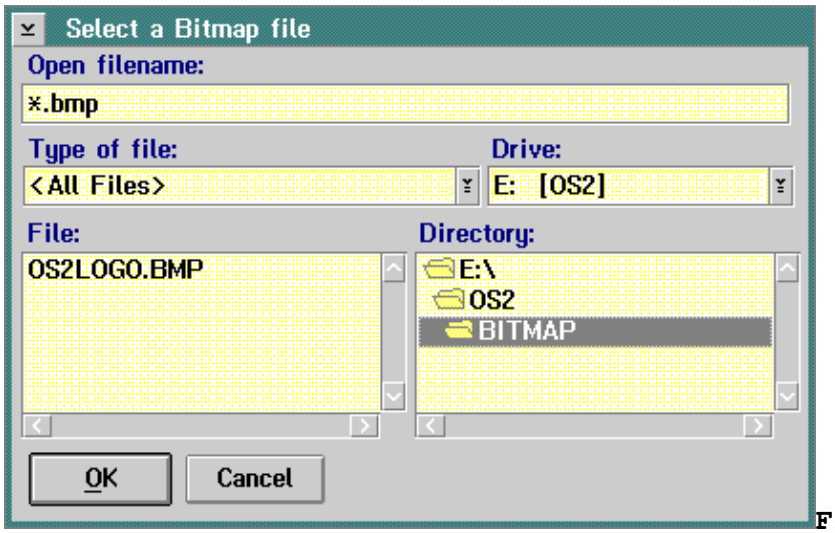


**Figure 70** Sample file dialog overriding default parameters

The initial type parameter lets you set the initial value displayed for **Type of file**. The default is all types. The file type is different from the file extension of the file name. The file type is saved in the file's extended attribute. File types in OS/2 let you create a special link between program objects and data file objects. When objects are linked, you can open a data file and a program at the same time. OS/2 defines a number of data types, such as Assembler Code, C Code, Executable and Icon. A program can also add its own data type . For more information about file types see the OS/2 online 'Master Help Index'.

Other parameters include the drive list and type list. These two parameters let you limit the values displayed for the drive and type fields. Both require compound variables. The file dialog can also allow for multiple file selection. A detailed description of the parameters is contained in the *Alphabetical Reference*. If you do not want to change a default parameter, omit it by leaving a blank between the commas.

## Font selection dialog

The font selection dialog (VRFontDialog) is used to obtain a font specification for use with the Font property and anywhere else a font string can be used.

An initial font can be specified as a parameter to VRFontDialog, as well as a title string:

```
font = VRFontDialog( VRWindow(), '10.Helv.Bold', ,
'Choose a font!' )
```

If the user selects a font, the string describing the font is returned. A null string is returned if the **Cancel** button is selected.

The font string is returned in the format described in the Font property section of the *Reference*.

## Message dialog

The message dialog VRMessage displays information. The message could be an error condition, descriptive information or a query. For example, the following code

```
call VRMessage VRWindow(), 'Welcome to the world of VX-REXX'
```

would display the message dialog in Figure 71.

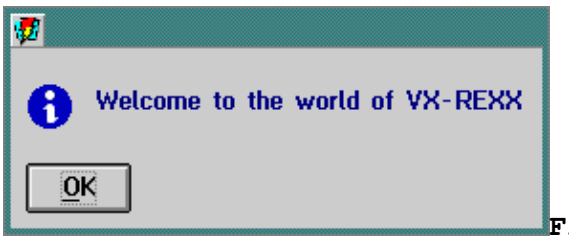


**Figure 71** Message dialog with default parameters

The two required parameters are the window to which the dialog is attached, and the message to be displayed. In this example the window is obtained using the function VRWindow and the message is 'Welcome to the world of VX-REXX' . Additional parameters let you change the title, add an icon, add buttons and set default and cancel buttons. For example, the following code would display the message shown in Figure 72. Note that a comma at the end of a line is the REXX line continuation character.

```
button.0 = 2
button.1 = 'OK'
button.2 = 'Cancel'
call VRMessage VRWindow(), 'Welcome to the world of VX-REXX', ,
'VX-REXX', 'I', 'button.', 1, 2
```


**Figure 72** Message dialog with user defined buttons

The third argument sets the title for the dialog to 'VX-REXX'. The fourth argument displays the information icon; you can choose from several icons such as a question mark, exclamation mark, or asterisk. A list of icons is included in the VRMessage description in the online *VX¨REXX Reference*.. The stem variable defines buttons to be displayed in the message box. Notice that the compound variable parameter has a period (.) at the end of the name. The last two arguments indicate which two buttons should be activated if the **Enter** or **Esc** keys are pressed. Pressing **Esc** or **Enter** will close the dialog.

## Multiline message dialog

The multiline message dialog VRMessageStem is used to display a message that requires more than one line. The arguments are the same as those for VRMessage except for the second argument which defines the text for the message dialog using a compound variable. For example, the following code would display the message window shown in Figure 73. Note that a comma at the end of a line is the REXX line continuation character.

```
message.0 = 3
message.1 = 'Version 1.00'
message.2 = 'Copyright (c) 1993 WATCOM International Corporation'
message.3 = 'WATCOM is a trademark of WATCOM International'
button.0 = 1
button.1 = 'OK'
call VRMessageStem VRWindow(), 'message.', ,
'VX-REXX - Product information', 'N', ,
'button.', 1, 1
```


**Figure 73** Multilinemessagedialog

This message displays the same information as the **Product information** item from the VX-REXX **Help** menu. Notice that the dialog box adjusts its size to fit the long lines.

The reference to the window object and the message are the only required parameters. The additional values set the title and add the buttons.

## Prompt dialog

The prompt dialog VRPrompt lets you display a simple input dialog to request data. The arguments for the function are similar to those for the message function. An additional parameter is provided for saving the user's response . For example, the following code will display the prompt window shown in Figure 74.

```
userName = ''
button.0 = 2
button.1 = 'OK'
button.2 = 'Cancel'
call VRPrompt VRWindow(), 'Please enter your user name:', ,
'userName', 'Logon', 'button.', 1, 2
if result = 2 then userName = ''
```

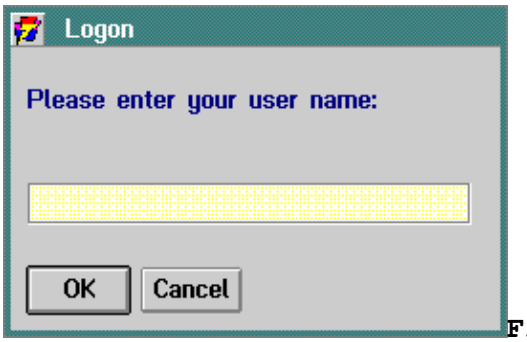


**Figure 74** Prompt dialog

In this example the value that the user enters is returned in the variable **userName**. There are two buttons: OK and Cancel. Pressing **Enter** is equivalent to clicking OK and pressing **Esc** is equivalent to clicking Cancel.

The value returned by VRPrompt indicates which button was pressed. In this example the program checks to see if Cancel was chosen (result = 2). If it was, any value entered is ignored.

# INI files

OS/2 stores configuration and profile data in INI files. VX-REXX provides several functions for manipulating those files: VRDelIni, VRGetIni, and VRSetIni .

INI files are used to efficiently store text and binary data. User preferences are often stored in INI files, for example. If possible, it is best for an application to use its own private INI file rather than one of the OS/2 system INI files . Ideally this file should exist in the same directory as the program executable . You can use the Application object in conjunction with the VRParseFileName function to determine where your executable is:

```
exePath = VRGet( 'Application', 'Program' )
iniDir = VRParseFileName( exePath, 'DP' )
iniFile = iniDir || '\myini.ini'
```

```
If the INI file does not exist or is missing data, your application
should create a new INI file with reasonable default values.
```

# File system manipulation

File system manipulation in REXX is limited to a few built-in functions and relies mostly on the use of commands to the CMD host environment. VX-REXX defines a complete set of file system functions for performing common tasks:

oFile and directory creation, deletion and copying: VRCopyFile, VRCreateFile, VRDeleteFile, VRFileExists, VRIsDir, VRMkDir, VRRenameFile, VRRmDir

oGetting and changing the current drive and/or directory: VRChDir, VRChDrive, VRCurrDir, VRCurrDrive, VRDir

oSetting file attributes: VRChAttr, VRFileDate

oRetrieving disk information: VRDiskInfo, VRDiskLabel, VRFindFile

oParsing and expanding file paths: VRExpandFileName, VRParseFileName

# Creating custom dialogs

In a previous chapter, you have seen how to use built-in dialogs. This chapter describes how to create your own custom dialogs using window files.

Window files behave the same way as the built-in dialog functions -- you call them, the window appears, the user closes the window, then the calling program continues. Windows that operate in this way are called *modal*. A modal window automatically disables its parent and any sibling windows. The calling window file is also suspended while the called window is active. When a modal window closes, it enables its parent and sibling windows, and execution of the calling window file continues. Your program should use a modal window whenever it has to get user input before continuing.

In contrast, a *modeless* window is part of the window file that loads it. A modeless window does not suspend the code that loaded it, nor does it disable any other windows. You can use modeless windows to allow your user to work with several windows at the same time. This type of window is covered in the ' Secondary windows' chapter, and is not discussed here.

Modal windows do not have to be separate window files. You can also create modal windows in the same file using the techniques discussed in the ' Secondary windows' chapter. This chapter, however, is only concerned with the case where a modal window is a separate window file.

The sections in this chapter take you through the process of creating a sample modal dialog which prompts the user for two pieces of data: a name and a password.

## Calling the modal window file

In this example, we will create a simple modal dialog that asks the user for some information. The dialog will be kept in a second window file. The first file calls the second file which returns a name and password.

When windows are saved in separate files, one window is activated from another window by calling it with the file name as in:

```
values = Window2( VRWindow() )
```

where **Window2** is the file name of the second window file. The VX-REXX function VRWindow, which returns the internal name of the current primary window, must be passed as the first argument. The second window file needs the name of the current window in order to disable it and all of its child windows (if any). The second window disables its parent automatically; you do not have to add any REXX code to do this.

On your main window add a push button, then modify the button's click event to the following:

```
PB_1_Click:
values = Window2( VRWindow() )
```

```
if values \= '' then do
parse var values name ',' password
say 'Name =' name ', Password = ' password
end
else do
say 'The dialog was canceled.'
end
return
```

```
This code assumes Window2 will return the name and password in a single
comma delimited string. If the user closes the window or presses the
cancel button, Window2 will return the null string.
```

## Creating a new window file

To create a new window file, choose the **New window file** item from the **Project** menu. When you do this, the current file is closed and a new window file is created and displayed. At the same time, the file list window will appear. This window is discussed in the 'Multiple file projects' chapter.

The new window is automatically given a unique file name of the form **Window** followed by a number; for example, **Window2**. To rename this window, choose the **Save as** item from the **File** menu. VX-REXX will then prompt you for the new name of the window file.

The name of the new file is added to the file list. If the section list window is open, you will see that sections have been generated for the window as they were for the first window -- Fini, Init, Main, Quit and Window2_Close.

Set the window's Caption property to **Logon**. Then add two entry fields, some descriptive text, and OK and Cancel buttons to the new window so that it resembles Figure 75.

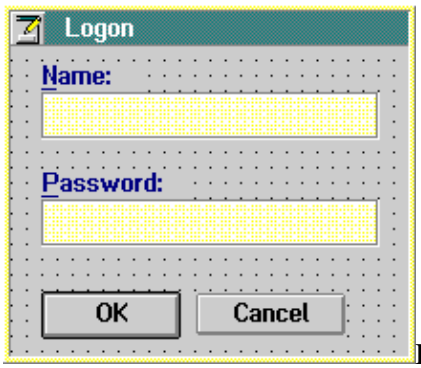


**Figure 75** A custom dialog

## Returning a value

Window2 returns a name and password in a single comma delimited string. The value returned by the Fini routine in Window2 is the value that is passed back to Window1. The Fini section is automatically generated

when you create a new window file. It returns 0 by default.

For our example, Fini returns the **Retcode** variable which we will set in the OK button's click routine. First, edit the PB_1_Click routine in Window2 to match the following code.

```
PB_1_Click:
name = VRGet( 'EF_1', 'Value' )
pswd = VRGet( 'EF_2', 'Value' )
Retcode = name || ',' || pswd
call Quit
return
```

The Click routine calls Quit to terminate the dialog.

**Note:** If you have changed the names of the entry fields, replace **EF_1** and **EF_2** with the actual names of the entry fields.

Next, modify the Click event for the cancel button to close the dialog window without changing **Retcode:**

```
PB_2_Click:
call Quit
return
```

Initialize the **Retcode** variable in the Init routine of Window2:

```
Init:
Retcode = ''
window = VRWindow()
call VRMethod window, 'CenterWindow'
call VRSet window, 'Visible', 1
call VRMethod window, 'Activate'
drop window
return
```

And finally, change the Fini routine to return the **Retcode** variable:

```
Fini:
window = VRWindow()
call VRSet window, 'Visible', 0
drop window
return Retcode
```

## Running the program

You are now ready to run the program. If the user closes the dialog by pressing the OK button, Window2 will return the contents of the name and password fields . Otherwise, it will return the null string.

## Multiple file considerations

Because modal windows are kept in different files, you should be aware of the following aspects of multiple file programming:

oThe scope of REXX variables and labels
oThe scope of object names

For programming details concerning multiple file projects, see the ' Multiple file projects' chapter later in this manual.

# Secondary windows

The previous chapter showed you how to create a custom dialog using a modal window in another file. Calling a modal window file is like calling a routine : the parent program waits and the parent window is disabled until the child window is closed.

There are times when this type of interaction is inappropriate, when you want a dialog window to be visible and enabled while the main window is still enabled . In this case, you want a *modeless window*. A modeless window does not disable its parent window when it is open. You may have already seen modeless windows in the VX-REXX design environment; the tool palette, section list, and file list windows are all modeless windows.

There are also times when you want a *modal* window to be part of the same file as the main window so that data can be easily shared between the two windows.

This chapter describes how to use windows which are part of the same file as the main window for both modeless and modal interactions.

## Primary vs. secondary

The first window you create in a file is the *primary window*. Normally the main user interaction takes place in this window. It presents information that is used independently from information in all other windows. Other windows that you create as part of the same file are *secondary windows*.

Secondary windows are used to supplement the interaction in the primary window. They are associated with the primary window in the following ways: Secondary windows are closed, minimized, or restored when the primary window is closed, minimized, or restored. You can also specify that a secondary window moves when the primary window is moved (see the MoveWithParent property).

When a window file is called, all of the code is loaded and the primary window is displayed. The secondary windows are loaded by the program when required . Because all of the code is always loaded, the REXX labels, variables and object names are known to all of the windows in the file. Moreover, when a secondary window is opened, the events associated with the primary window can also be used. The action in the primary window is not suspended.

The following sections show you how to create and use modeless and modal secondary windows.

## Window list window

To create a secondary window, use the window list window. The window list provides commands to create, open, close and delete secondary windows. To display the window list, choose **Window list** from the **Windows** menu. It will be as shown in Figure 76.

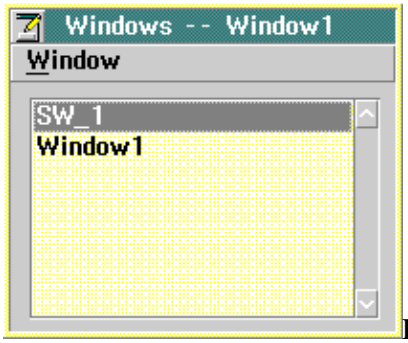


**Figure 76** Window list window

The window list displays the names of the primary and secondary windows in the current file. In this example there are two windows: the primary window, **Window1**, and a secondary window, **SW_1**. To see the list of windows in a different file, double click on the file name in the file list window.

The **Window** menu contains commands to help you maintain multiple windows in a file. These commands are described by task in the following sections.

## Creating and editing a secondary window

To create a secondary window, choose **New window** from the **Window** menu in the window list and enter a name for the window. Doing this creates a new empty window . The current window does not close so you can align and resize your windows as necessary. If you want to close a window, click on its system menu, or select the window from the window list and choose **Close** from the **Window** menu. The **Open** command will open a window for editing.

When a secondary window is created, a unique window name is generated for it . By default the name is **SW_x**, where **x** is some number. For example, the name might be **SW_1** if this were the first secondary window. You can specify a new name when the window is created or change the window name by changing the Name field in the window's property notebook. If you are creating several windows, you may want to change the default names to make it easier to remember their function .

If the file list window is open, you will notice that adding secondary windows does not change it. You are not creating a new file. However, you will see new sections called SW_1_Close, SW_1_Create, SW_1_Init and SW_1_Fini in the section list. When you add event code for an object in the new window, the section name for the event will also be added to the section list. You can use the **Section** menu in the section list window to manage the code for any of the windows in the file.

Once the new window is created, you can add objects and change properties as you would for any other window. There are several window-specific properties you might like to set: BorderType, TitleBar, HideButton, MaximizeButton, MinimizeButton, MoveWithParent and SystemMenu.

MaximizeButton, HideButton and MinimizeButton are probably not needed on a secondary window. If the primary window contains these buttons, a secondary window will be minimized or restored along with the primary window.

MoveWithParent should usually be set so that if you move the primary window the other window moves with it.

The BorderType property on a secondary window that is not meant to be sizeable should be set to either **Thin** or **Dialog**.

Many applications include a system menu as a standard way to close a window . If you want to be able to close a secondary window in this way, select the SystemMenu property on the Frame page of the window's property notebook.

## Opening a secondary window at run time

When a file contains multiple windows, you must explicitly load the secondary windows in order to open them. You load a window with the VX-REXX function VRLoadSecondary. How you call VRLoadSecondary determines whether the secondary window is loaded as a modal window or as a modeless window. For example, the following statement loads a modeless secondary window named SW_1:

```
window = VRLoadSecondary( 'SW_1' )
```

VRLoadSecondary returns the internal name of the loaded window. If it cannot load the modeless window it returns an empty string. Execution immediately continues with the next REXX statement in the section.

In contrast, the following statement loads SW_1 as a modal secondary window :

```
call VRLoadSecondary 'SW_1', 'Wait'
```

Unlike the modeless case, the call to VRLoadSecondary does not return immediately, but rather disables the parent window and processes events for the secondary window. The call to VRLoadSecondary returns only when the secondary window is destroyed. It is similar to calling a modal window in another file, except that the primary window and secondary window are actually in the same file. Unlike modal window files, no arguments can be passed to the secondary window and no value is returned, so communication between the primary and secondary window is done using global variables.

You should use the REXX **procedure** statement on all the event routines of modal secondary windows. If you do not, variables in the routine which loaded the window may have their values changed. This is because the loading routine will indirectly call the event routine via VRLoadSecondary. As a result, variables in the loading routine will be exposed to the event routine unless the **procedure** keyword is used.

Loading a modal secondary window also sets the value returned by VRWindow to be the internal name of the secondary window and resets it when the window is destroyed. The value of VRWindow is not changed when a modeless secondary window is loaded.

When a secondary window is loaded, a Create event occurs. By default, the _Create event calls the window's Init routine (for example, SW_1_Init) to center the secondary window over its parent and make it visible and active. You can modify this behaviour by changing the window's Init routine. Modal secondary windows should always be made visible, however, because the parent window will have been disabled.

Because they are in the same file, primary and secondary windows can easily share information through variables. Variables in a file can be used in any section or window in that file.

If you want a secondary window to be opened with the primary window, you may want to load it in the Init section so that it is opened when your application is started.

## Closing a secondary window at run time

Secondary windows remain open until you close them. If the secondary window has a system menu, you do not have to add any code to handle the Close event; VX-REXX automatically generates REXX instructions to close the window. However , if you provide a push button or menu item to close the window, you must add code to that object's click event to close the window. To close a secondary window, call the Fini routine for the window. For example, suppose you want to close a window called SW_1 when push button PB_1 is clicked. The Click event for the push button would look similar to:

```
PB_1_Click:
call SW_1_Fini
return
```

If you destroy a modeless window you must use VRLoadSecondary again to reopen it.

If you wanted to hide a modeless window rather than destroying it, you could replace the code in the Fini routine with the following:

```
call VRSet 'SW_1', 'Visible', 0
```

If you make the window invisible, you can open it again by setting the window 's visible property to 1. Note that it is faster to change a window's visibility than to destroy and load it again. Do not make a modal secondary window invisible, however, as the parent window will still be disabled.

## Saving a multiple window file

All of the windows in a file are saved when you save the window file using **Save** or **Save as** from the file list window, or when you save the project using the **Project** menu.

## Deleting a window

Use **Delete** from the window list **Window** menu to remove a window from a file . When you choose **Delete**, you are asked to confirm the deletion. If you proceed, the window and any event sections associated with that window object are removed from the file. Once you delete a window, you cannot add it back. You have to recreate it.

# Multiple file projects

This chapter describes the tools that help you create and maintain multiple file projects. As you develop more complex applications you may require more than one file. Additional files may be simply code (for example, to provide general routines to share between windows or projects) or may be additional windows such as dialogs to display messages or request information.

## Overview

A VX-REXX project consists of a project file and one or more window and/or code files. A window file consists of REXX code that describes the actions to perform and data that describes one or more windows. A code file is simply REXX code; it does not have a window associated with it. A VX-REXX project can consist of code files only.

A window file or code file is a REXX program file. Each file has a separate name space. This means that labels, variables and object names are local to that file. To exchange data between files, values must be passed as parameters when calling a file, or returned as a single return value. VX-REXX also provides a global variable facility that can be accessed using the PutVar and GetVar methods.

The project file lists all of the window and code files associated with the project. All of the files are often located in the same directory, although this is not necessary. When you create multiple file projects you can include files from other projects or general code files that are shared among projects. These files might be located on a network. However, when you make an executable file for a completed project, all of the code is assembled into one executable file.

VX-REXX provides an integrated project management facility that simplifies the task of creating and organizing multiple files in a project. When your project requires more than one file, use the Files window to create, edit and save additional files. The following sections describe the Files window and how to use it to maintain a multiple file project.

## File list window

The file list window provides commands to manage multiple file projects. You can create new window files, add or remove files from within a project, and save existing files. To display the file list window, choose **File list** from the **Windows** menu. It will be similar to the window shown in Figure 77.

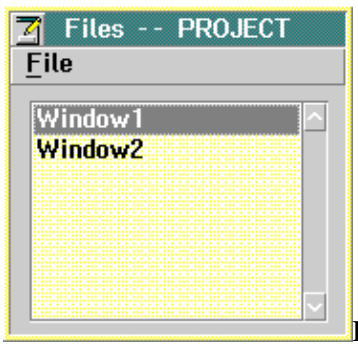


**Figure 77**
File list window

The list box displays the names of all files in the project. In this example, there are two window files called Window1 and Window2. The **File** menu contains commands to help you maintain a multiple file project. Each of these commands is described in the following sections.

## Creating and editing a new file

Both the **New code file** and **New window file** items in the **File** menu add a new file to a project. A new code file lets you create REXX routines that are not associated with a particular window but that could be called by any window in any project . This allows you to develop general routines that can be shared across projects . A new window file creates a new window with its associated code. This window can be called by another window in this project and can also be added to another project.

When you choose **New code file** the current file is closed and a new empty file is created. A unique file name is generated as **File** followed by a number; for example, **File1**. The name is added to the file list. Two new sections, **Main** and **Halt**, are generated when the file is created. The section editor is opened and the Main section is loaded. It only contains the section header, routine name and a return statement as in the following:

```
/*:VRX */
Main:
signal on halt

return
```

You can add all your code to this section or you can create new sections using the **New** menu item from the **Section** menu in the section list window.

When you choose **New window file**, the current file is closed, a new window file is created and an empty window is displayed. Choosing this menu item is identical to choosing the **New window file** item in the VX-REXX **Project** menu. A unique file name is generated as **Window** followed by a number; for example, **Window2**. The new name is added to the file list. If the section list window is open, you will see that sections have been generated for the window as they were for the first window (Fini, Init, Main, Quit and Window2_Close).

Note that if you select **New code file** or **New window file** when a new project has just been created (i.e. Window1 is still empty) Window1 will be removed before the new window or code file is created. This feature can be used to create a project that contains only code files.

Edit your new window in the same way as other windows by adding objects, modifying properties and adding event code to the objects or adding general code.

Only one code or window file can be open for section editing at one time. When you create a new code file or window file the current file is closed. You can open an existing code file or window file by double-clicking on its name in the file list or by clicking on the name to select it and then choosing **Open** from the **File** menu. When you do, the current file is closed and the selected file is opened.

As you work with larger projects, you may find it useful to edit all of the sections in a file at once. This allows you to easily make global changes and to view or print all of the code at once. When you select a file and then choose **Edit** from the **File** menu in the file list window, the external editor will load all the code for the selected file; all sections of the file will be displayed. If the selected file is the file currently open for section editing, it will be closed before starting the external editor to help prevent you from editing the same code in two different windows.

You can use the external editor to edit as many files at once as you like . This makes it easy to compare and copy code between files. However, VX-REXX does not keep track of your external editor sessions. You must remember to save the edited file before saving or running a project or the old version of the code will be used. You should also be careful to avoid editing the same file more than once at the same time, and to avoid opening a file for section editing when you already have it in an external editor.

# Calling a file

One of the files is specified as the main file; it is the file that is executed when the application is started. The main file can be a window file or a code file. Initially the main file is the first window that is created when the project is created (Window1). You can change the default main file by changing the file name specified for **Main file** in the **Run options** dialog accessed from the **Options** menu.

Other files are called from this file and those files could call additional files, and so on. Files are called as routines: they can be called as procedures or as functions, and they can be passed arguments and return values as shown in the following sections. When a file is called, any action in the calling file is suspended until the called file returns. This means that when a window in a second file is started, you cannot process any events in the first window until the second window is closed.

## Calling a code file

Activating a code file from another file is the same as calling an external subroutine in REXX. You call it by the file name as in:

```
call File1
```

where File1 is the name of the code file.

If the code file requires information from the calling file, it can be passed as arguments. For example:

```
call File1 parm1, 'hello'
```

passes the value of the variable **parm1** and the string 'hello' to the code file File1. The values of the arguments are retrieved in File1 as they would be in any REXX program file. It is up to you to assign the arguments to variables with the **arg** or **parse arg** instructions. For example, you may assign the arguments from the previous example to variables by adding the following statements to the Main section of File1.

```
parse arg parm, string
```

The value of **parm1** would be assigned to the variable **parm** and 'hello' would be assigned to the variable **string**.

## Calling a window file

When windows are saved in separate files, one window is activated from another window by calling it with the file name as in:

```
call Window2 VRWindow()
```

where Window2 is the file name of the second window.

The VX-REXX function VRWindow returns the internal name of the current primary window, which must be included as the first argument. This reference is passed to the called window file to identify the calling window. If this parameter is not passed, the calling window will appear to be enabled. If you try to execute an event in the calling window while the called window is displayed, the events are queued and later executed when the called window is closed. This is generally not desirable.

Because these windows do not share code, any information that is shared by the windows must be passed between the windows. Data can be passed to a window by including arguments in the call to the window as in the statement:

```
call Window2 VRWindow(), parm1, 'Hello'
```

In this example, the value of the variable **parm1** and the string 'Hello ' are passed to Window2. When the second window is started the arguments can be retrieved. The arguments are stored in the compound variable **InitArgs**. **InitArgs.0** contains the number of arguments. The values of the arguments are saved in **InitArgs.1**, **InitArgs.2**, and so on. In this example, **InitArgs .1** is set to the value of **parm1** and **InitArgs.2** is set to the string ' Hello'. The argument values can be used in any section of code in Window2 . The value of the argument VRWindow() is not included in the compound variable .

## Returning a value

A called file may need to return a value to the calling file. For example , a second window may present a dialog which requests information that is needed by the calling window. A single value can be returned to the calling window by modifying the return value of the return statement in the Fini section of the called window. The Fini section is automatically generated when you create a new window . It returns 0 by default. For example, you could return a value as in the following:

```
returnValue = 'goodbye'
return returnValue
```

The code in the calling window would be similar to the following:

```
returnValue = Window2( VRWindow(), parm1, 'Hello' )
```

The variable **returnValue** contains the string 'goodbye'. If you had several values to return you could join them together in one string and then parse the return value. However, VX-REXX provides an alternative to make it easier to return multiple values.

### Returning multiple values

All VX-REXX projects have an invisible application object that you can use to pass results between files. The object has two methods, PutVar and GetVar, that store and retrieve the value of a variable for the application object. This provides a global variable facility for VX-REXX applications.

For example, suppose code in Window1 calls a routine in Window2. The routine must pass back two values to Window1. The calling code in Window1 looks like this:

```
call Window2
call VRMethod 'Application', 'GetVar', 'parm.'
say 'Window2 returned' parm.1 'and' parm.2
```

and the called routine in Window2 looks like this:

```
main:
parm.0 = 2
parm.1 = parm1
parm.2 = 'Hello'
call VRMethod 'Application', 'PutVar', 'parm.'
return
```

The arguments **parm1** and **'hello'** are stored in the compound variable **parm.** This variable is stored in the application object when Window2 executes the PutVar method. The variable becomes available to Window1 as soon as the GetVar method is called.

You can use PutVar in the Fini section to store the values of a stem array before the file exits. The calling file then uses GetVar to retrieve the values .

# Saving files

You may be familiar with saving projects with only one window -- you entered a name for the window file and a name for the project file. When you save a project with multiple windows you are asked for a file name for each window and code file. By default they are set to Window1, Window2... and File1, File2 and so on. The project file contains all of the window and code file names associated with the project.

If you have many windows or files, you may want to change the default file names so that it is easier to remember their function. Note that if you change the file name and you have code that refers to the file, you must also change the name in any call statement that refers to it.

## Saving one file

When you are working with multiple file applications, you may need to edit only one window or code file, or you may want to checkpoint your changes before editing another window. To save changes for one file you do not need to save all of the project files. You can save the window or code file with the **Save** or **Save as** menu items from the **File** menu in the file list window.

If you choose **Save** and it is the first time you are saving the file, you will be prompted for a file name and location. Otherwise, you will not be prompted to enter a name. The file will be saved with its current name and the previous contents of the file will be overwritten.

If you want to save the file with a new name, choose **Save as** from the **Project** menu. When you enter a new file name, a file is created with that name and the file list is changed. The file with the old name (e.g. Window1) still exists but the file remains unchanged and is no longer part of the project.

## Removing or adding a file

**Remove** in the **File** menu lets you remove a file from a project. When you choose **Remove**, the window file or code file is removed from the project file but is not deleted from the disk.

The **Add** menu item in the **File** menu lets you add an existing window file or code file to your file list. You may want to use this feature to include a window from another project, to share code on a network, or to add a file that has been removed.

When you add a file, only a reference to that file is added to the current project. If the file is shared between projects, changing it in one project changes it in both. However, when you make an executable for the current project, the code for the referenced file is included in the executable file.

## Saving a file as text

Since window descriptions are saved in a binary format you cannot change them directly. However, **Save as text** in the **File** menu lets you save a window file or code file as a text file. It has the same name as the file but has the extension **VRT** . A window file saved as text contains the code that defines the window as well as the REXX code. You can edit and print this file with any text editor.

## Loading a file

The **Load** menu item in the **File** menu lets you load several different types of files into a VX-REXX project. You can load a file that was saved with the **Save as text** command. This lets you edit a project outside of VX-REXX and then load it back, converting it to VX-REXX internal format.

You can use **Load** to copy a file from another project. Whereas **Add** only adds a reference to a file, **Load** copies the file into a new file in the project. You can change the name when you save the file or project.

Load can also be used to load a REXX file. If you load a file with an extension other than **VRX** or **VRT** it is assumed to be a REXX file.

# Adding help to a program

This chapter shows you how to add online help to an application. It is good programming practice to include online help in all applications. Three types of online help are supported by VX-REXX:

oInformation Presentation Facility (IPF) help files. An IPF help file ( .HLP) is a binary file generated by a help compiler from a set of tagged text files. The IPF help compiler is part of the Client/Server Edition of VX-REXX and the OS/2 Toolkit, but is not included with the Standard Edition of VX-REXX.

oSimple text files. If you do not have access to an IPF help compiler, you may use the built-in text file facility instead. The text will be displayed in windows similar to those used by the IPF help system.

oCustom help files. Help requests may be intercepted and handled directly by your project using the Help event.

The user invokes help by pressing the **F1** key at any time or by selecting an item from the help menu.

VX-REXX also supports an optional status area on windows where hints -- single-line messages -- are displayed. Hints are updated automatically whenever the user moves the pointer over an object. For more information on hints, see the section on using the Window object in 'Using objects'.

## Using IPF help files

### Setting the file and title

To associate an IPF help file with a window, set the window's HelpFile property:

```
call VRSet 'myWindow', 'HelpFile', 'myhelp.hlp'
```

```
If you do not give an absolute path and the file is not found in the
current directory, the paths in the HELP environment variable will be
searched.

The title for the IPF help is set using the window's HelpTitle property :

call VRSet 'myWindow', 'HelpTitle', 'Help for my application'
```

### Help tags

Once the IPF file is associated with the window, set the HelpTag properties of the window and the objects on the window to the appropriate IPF tag values. The tag value is a number that was defined when the help file was created. It specifies which section of the help file to display when the **F1** key is pressed.

The following instructions set the help tag of the window to 1000, and sets the tags for two push buttons to 1001 and 1002:

```
call VRSet 'mywindow', 'HelpTag', 1000
call VRSet 'PB_1', 'HelpTag', 1001
call VRSet 'PB_2', 'HelpTag', 1002
```

If the **F1** key is pressed but no HelpTag property is defined, the parent object's help is invoked.

See the section for the HelpTag property for information on how the special tags **help index**, **help contents**, **extended help**, **general help** and **using help**.

### Notes on IPF tags

Help tags correspond to the **res**, **name** or **id** attributes of a heading in an IPF file. The **res** attribute must be a whole number in the range 1 to 64000, but **id** and **name** may be any series of alphanumeric characters. For example:

```
:H1 res=1000.This is a heading
```

Links may be established from one help file to another help file (.HLP) or online book (.INF), or from one online book to another online book (but not to a help file), but the headings to be linked must be made **global** and only the **res** attribute can be used:

```
:H1 global res=1000.A chapter
```

To link to this heading in the 'test.hlp' help file, use:

```
:LINK reftype=hd database='test.hlp' refid=1000.A chapter
```

If you wish to use symbolic tags (**id** or **name**) instead of or in addition to numeric tags (**ref**), the **global** attribute cannot be specified and hence no external links may be made to that heading.

For more information, please consult the documentation for the OS/2 help compiler.

## Using text files for help

The HelpText property may be used as an alternative to the IPF help facility . You may set the help text for an object directly as follows:

```
text = 'This is my sample help text.'
call VRSet object, 'HelpText', text
```

Long text will be word wrapped. You can force new lines by embedding carriage return-line feed pairs ('0d0a'x).

The HelpText property can also display text defined in a file:

```
call VRSet object, 'HelpText', '(myhelp.txt)'
```

The name of the help file is enclosed in parentheses. If an absolute path is not specified, the file must exist in the current directory.

The title of the help window is defined by setting the HelpTitle property on the window object. All objects on a window use the same help title.

As with IPF help, if the **F1** key is pressed but no HelpText property is defined, the parent object's help is invoked.

## Invoking help directly

Help can be invoked under program control by using the InvokeHelp method on any object:

```
call VRMethod object, 'InvokeHelp'
```

The help will be displayed as if the user had pressed the **F1** key.

The InvokeHelp method can be used in the implementation of a help menu. After defining the appropriate menu items with the menu editor, define the Click event for each menu item as follows:

```
Menu Click:
obj = VRInfo( 'object' )
call VRMethod obj, 'InvokeHelp'
return
```

Now simply set the appropriate help properties on the menu items in the Init section.

# Custom help

If your help requirements are more complex, a Help event is available on each window. If the Help event is defined, pressing the **F1** key on any object on the window invokes the Help event instead of displaying help or text files as described above. In the event routine, VRInfo can be used to obtain the internal name of the object for which help was requested:

```
Window Help:
help_for = VRInfo( 'Source' )
```

For example, the InvokeHelp routine may be invoked:

```
Window Help:
help_for = VRInfo( 'Source' )
call VRMethod help_for, 'InvokeHelp'
return
```

A useful way to use the Help event is to display information from online books. An online book (.INF) is similar to a help file (.HLP) -- in fact, both are prepared using tagged text files and the same help compiler -- but is viewed using the OS/2 **view** command. To invoke **view** from your project, use the following syntax:

```
address cmd 'view' books topic
```

The *books* variable should hold the name of the book or books to be viewed and *topic* is the topic (as listed in the table of contents) to view. Several books can be combined into a single volume by using '+' signs, as in:

```
books = 'a2z+progguid+rexx'
```

If the topic is omitted, the contents page of the book is displayed instead .

For example, to display the online help for the VX-REXX Window object, use :

```
books = 'a2z'
topic = 'Window object'
address cmd 'view' books topic
```

By storing the book topic in an objects's HelpText property, pages from online books may be easily displayed when **F1** is pressed:

```
Window Help:
help_for = VRInfo( 'Source' )
books = 'rexx'
topic = VRGet( help_for, 'HelpText' )
address cmd 'view' books topic
return
```

# Debugging a project

VX-REXX provides integrated debugging tools within its development environment. The first level of debugging support provides handling and notification of run time exceptions. The second level provides a fully interactive program debugging environment. A traditional method of debugging REXX programs through the use of the **say** and **trace** instructions is also supported.

## Run time program exceptions

In the VX-REXX system, when a REXX program detects a run time error, the program is terminated and control is returned to VX-REXX. VX-REXX displays a window which notifies you of the error and where the error occurred within the program. Figure 78 shows a sample program exception window.

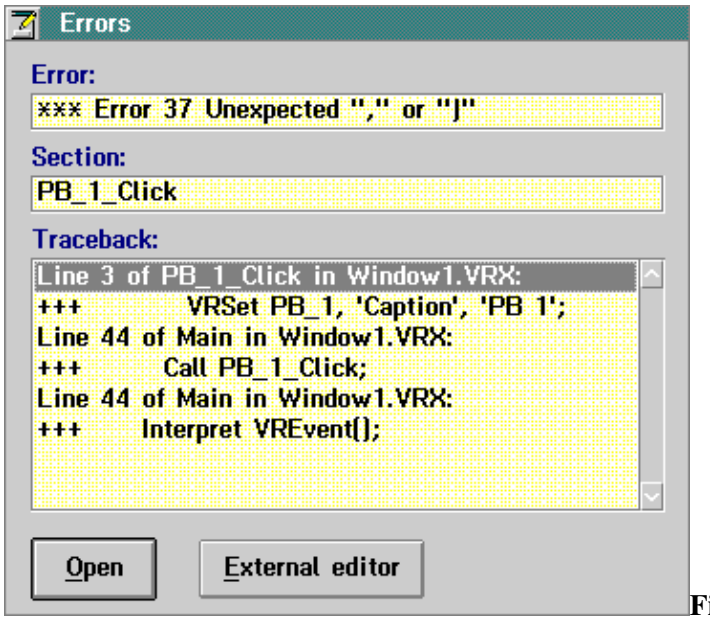


**Figure 78** Run time exception error handling

The error field displays the error which caused the program exception. Below that, the section field shows the section name of the selected line in the traceback list.

The traceback list shows the sequence of instructions that led up to the error. The top line shows the file, section, and line number of where the error occurred. The second line shows the erroneous REXX line itself. The next pair of lines identify the code that called the section with the error. Subsequent pairs of lines show which section called that section, and so on.

You can edit the section for any of the lines in the traceback list. There are two ways of doing this. Either double click on the line in the traceback list, or select the line, then press **Open** or **External Editor**.

You cannot continue a program which has terminated due to a run time exception; you must run it again from the beginning.

# The interactive debugger

A program is executed in VX-REXX using the **Run project** command. In this environment, program errors are handled as described previously. A program also can be run in a debugging environment which lets you trace program execution, set breakpoints, and display and modify variables. To use these features, use the **Debug project** command in the **Run** menu. No special preparation needs to be done to use the debugger.

## Debugger windows

There are four types of debugger windows: the section list window, source windows, the variable display window, and the results window. The debugger windows are all sizeable so that you can adjust them to your viewing preference. Each window type is described in the following sections.

### The section list window

The section list shows all of the files in the project, and the section names for the currently selected file. The section list is used to open additional windows to view the source for sections of REXX routines. A debugging session has only one section list.

If your project is made up of more than one file, the section list window will present a list of files, and a list of the sections in the files.

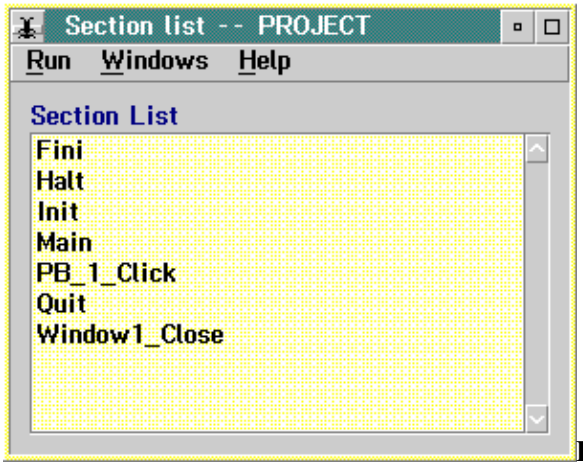


**Figure 79**
The section list window

### Source windows

A source window displays a section of the project. You can open multiple source windows, each viewing a different section. From a source window, you can set breakpoints and trace statement execution. You can also select variable names to be displayed in the variable window.

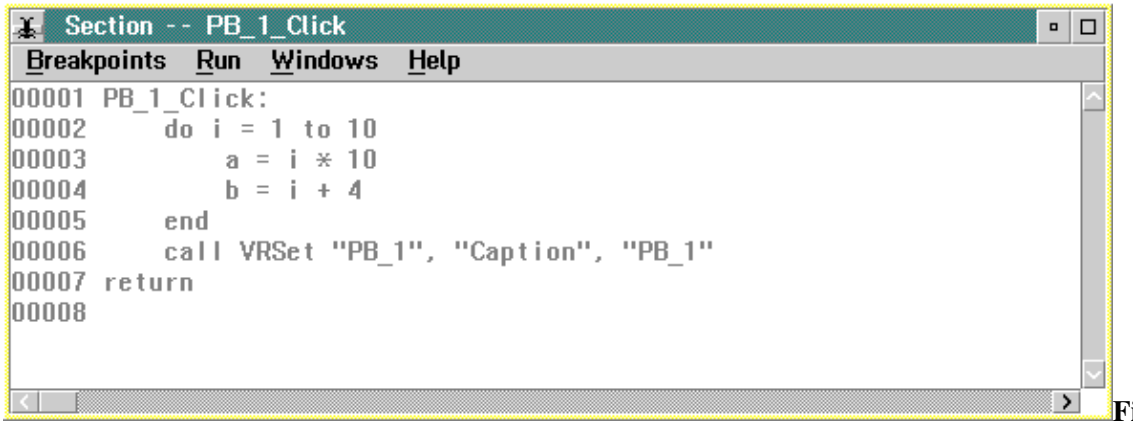


**Figure 80** A debugger source window

### The variable window

The variable window displays the values of selected variables. The window is updated each time control is returned to the debugger from the executing program. You add variables to the window by double-clicking on the variable name in any source window. The value of a variable can be changed by modifying the value displayed for that variable in the window.

In a multiple file application, the variable window only shows variables associated with the currently active file.

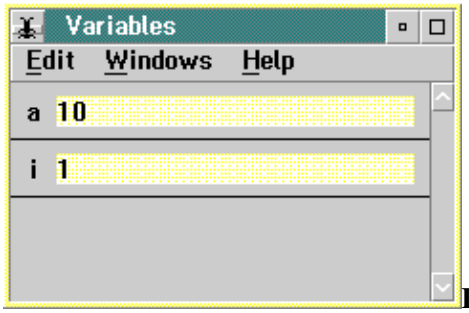


**Figure 81** The debugger variable window

### The result window

The result window displays the information that is output from the REXX interpreter. Three types of information are displayed: the REXX source statement that was run, the result of the statement that was run, and trace messages. The results window has an extra menu: the **Options** menu. The options menu is used to select the type of information to be displayed. The result window also allows you to enter and run simple REXX statements.

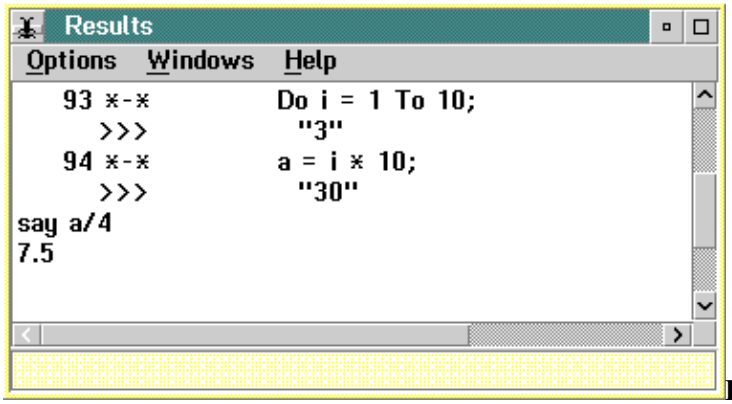


**Figure 82** The debugger result window

## Starting the debugger

The following steps would be used to start the debugger.

1.Open the project file to be run.

2.Choose **Debug project** from the **Run** menu.

When the debugger starts up, it displays the section list. The program has not started running at this point. You can examine your program, set breakpoints, and view variables before the program is run. A sample section list window is shown in Figure 79.

## Viewing REXX instructions

You can view any section from the list of names shown in the section window . To view a particular section, double-click on its name in the section list. In a multiple file application, you need to click on the file name before selecting the section name. This opens a new debugger source window to view the routine. An example source window is shown in Figure 80. The window title contains the name of the section. You can open multiple debugger source windows at the same time.

In addition to viewing the REXX statements in a section, a source window is used to set breakpoints, select variables to display, and control tracing of a program.

## Using breakpoints

In VX-REXX, you use breakpoints to interrupt your program after specified statements. One or more breakpoints can be set in a program. There are two ways to set breakpoints in a program:

1.Open a debugger source window for the section containing the statement .

2.Double-click on the line number for the statement.

or

1.Select **Set** from the **Breakpoints** menu in any debugger source window.

2.Select the file name and section name.

3.Type in the line number of the statement.

4.Click the **OK** button.

The line number of a statement with a breakpoint has a red background. Figure 83 shows a routine with two breakpoints set.

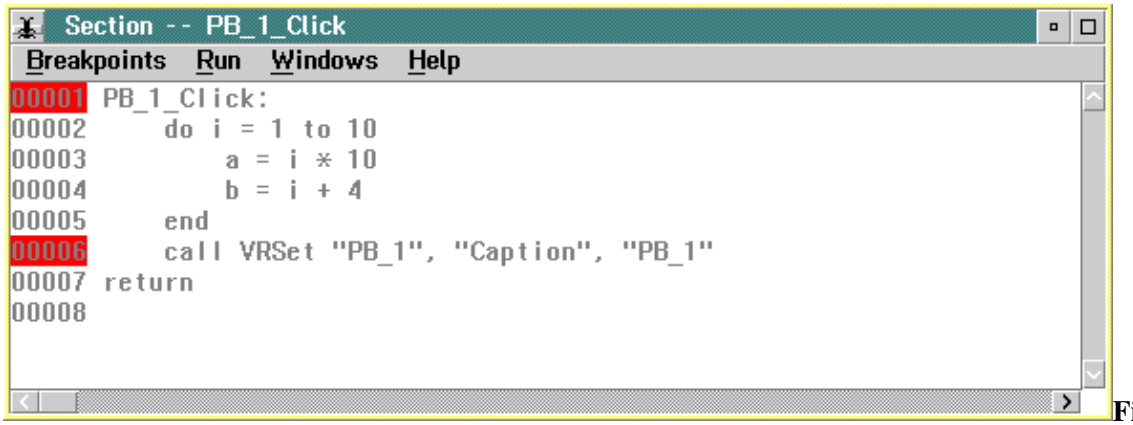


**Figure 83**
Source window with
breakpoints

Once the breakpoints have been set, you can run the program by selecting **Run** from the **Run** menu. The program runs normally until an instruction with a breakpoint is executed. At this point, the debugger suspends the program and opens a source window displaying the section which contains the breakpoint. In it, the instruction that was just executed is highlighted. The source window will appear similar to Figure 84.

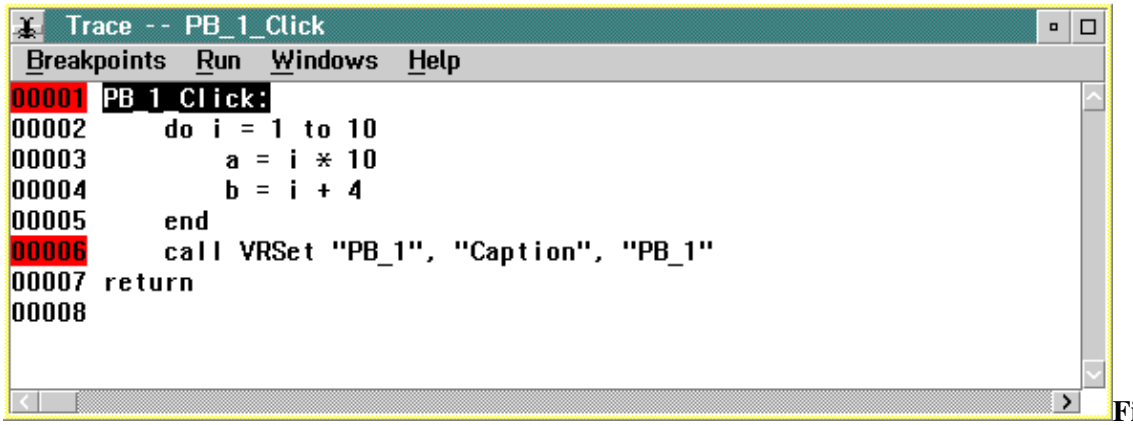


**Figure 84**
Program halted at a breakpoint

While the program is suspended at a breakpoint, you can add new breakpoints, clear existing breakpoints, view contents of variables, modify contents of variables, and resume running.

To clear a breakpoint from a statement, use either of the following procedures:

1.Open a debugger source window for the section containing the statement .

2.Double-click on the line number for the statement. This action clears the breakpoint.

or

1.Select **Clear** from the **Breakpoints** menu.

2.Select one or more breakpoints in the list. You can select all breakpoints by clicking the **Select All** button.

3.Click the **OK** button. The breakpoint will be cleared.

## Tracing a program

The debugger lets you interactively step through your program statement by statement. Tracing through your program is controlled by commands in the **Run** menu .

To run the next instruction in your program, select **Step** from the **Run** menu . After the instruction completes, control is returned to the debugger. Every time you execute a step, the variable list is updated, and the instruction you just executed is highlighted.

The **Step** command always steps into called routines. If you do not want to trace into a called routine, use the **Step Over** command. This command executes the called routine, then stops after the routine returns. This capability is useful to avoid tracing routines that you know work correctly. You can use **Step Over** at any time; if the next instruction is not a **call** instruction, **Step Over** just does a **Step** .

If you want to execute the same instruction again, use the **Redo** command from the **Run** menu. For instance, you might modify the contents of a variable which is used in the statement that was just run. Re-running the statement would use the new value.

To resume normal execution of your program, select **Run** from the **Run** menu. Your program will resume execution until another breakpoint is encountered, you interrupt the program, or your program completes normally. If your program completes or if an error occurs, the debugging session ends and you return to program editing. Interrupting a running program is discussed in a later section.

## Displaying and modifying variables

You can examine and modify the contents of variables while the program is being debugged. The debugger variable window displays a list of selected variable names with associated values. The values for these variables are updated each time that the debugger resumes control of the executing program. A variable window is shown in Figure 81.

To add a variable to the debugger variable window:

1.Open a debugger source window with a section that references the variable.

2.Double-click on the variable name.

The selected variable is inserted into the variables window. Its value is displayed in an entry field to the right of the variable name. If no value has been assigned to the variable, '(uninitialized)' appears beside the name.

Selection of a variable name is not context sensitive. Any word which is selected in a REXX statement will be used as a variable name, whether or not the word represents an actual variable name.

You can change the value of a variable before execution of the program resumes. To change the current value of a variable:

1.Add the variable name to the variable window.

2.Click on the entry field which contains the variable's value. A cursor will appear in the entry field.

3.Change the value. This new value will be used when the program continues running.

You can remove variables from the variable window once you are through with them. To remove a variable:

1.Click on the variable name in the variables window.

2.Select **Delete** from the **Edit** menu in the variables window. This action removes the variable from the window.

## Interrupting a program

Sometimes you may want to interrupt your program before it reaches a breakpoint or the end of the program. You can do this by selecting **Break** from the **Run** menu. You can pick this menu item whenever your program is running. After you interrupt the program, the debugger updates the trace and variable windows, giving you control again.

If you want to quit the debugger before your program terminates, close the section list window. You can do this at any time. After leaving the debugger, you will return to the VX-REXX design environment.

## Executing a REXX instruction

The project source cannot be modified while the program is being executed by the debugger. However, simple REXX statements can be entered and executed immediately in the debugger results window.

Suppose you wanted to print the value of some expression while debugging a program. If the expression was

```
interest * capital
```

```
You would do the following:
```

1.Select **Results** from the **Windows** menu, or type **Ctrl+R** to open the result window.

2.In the entry field at the bottom of the window, type the REXX instruction

```
say interest * capital
```

3.Press **Enter** to run the statement.

The output from the **say** instruction is displayed in the body of the results window.

### Run time exception handling in the debugger

A run time exception can occur while the program is executed under control of the debugger. When a run time exception is encountered, the debugger terminates and you return to the VX-REXX editing environment. You will see the Errors window that is described at the beginning of this chapter.

### Quitting the debugger

When your program terminates, the debugger shuts down automatically, and you return to the design environment. If you want to quit your application before it ends, close the section list.

## Debugging with the say and trace instructions

The REXX **say** instruction can be an effective way to trace your program and to display the value of variables. When the first **say** instruction is executed, the standard input/output (STDIO) console window is opened for your application. Results from the **say** and **trace** instructions are displayed in this window. If your program uses the **parse pull** instruction, it gets input from the STDIO window. The window remains open until your program terminates.

Adding conditional statements around **say** instructions allows the debugging statements to be executed only when it is desired. For instance, in the following example the **say** instruction would only be executed if the value of the variable **Debug** is true.

```
if Debug then do
Say 'Insert appropriate information to say'
end
```

The value of **Debug** can be set under program control or interactively using the debugger variable window.

Similarly, the REXX **trace** instruction can also be used to trace the execution of your program. The results of the trace instruction are displayed in the STDIO window. In this way, you can debug your programs as if you were using a generic REXX environment.

You can use the **trace** instruction along with the **Run project** command, but not with the **Debug project** command.

# Extending VX-REXX

This chapter discusses how to extend your VX-REXX projects using VX-REXX object libraries and REXX external function libraries.

## Object libraries

New objects can be developed for VX-REXX by third party developers and packaged as *object libraries*. After installing an object library using the instructions provided by the library developer, the objects in the library can be loaded into the VX-REXX design environment and used in your projects. Object libraries have extensions of .VXO or .DLL.

To load an object library, select the **Object libraries** item of the **Options** menu and add the name of the object library to the list of libraries to load. You may also set an option to automatically search for and load all installed object libraries.

After the object library is loaded, the tool palette is enlarged to accommodate the new objects. You may use the object as if it were one of the standard VX-REXX objects.

Projects that use an object from an object library will automatically search for and load the object library when they are run.

For more information on how to develop objects for use with VX-REXX, contact Watcom International.

## Function libraries

An external function library is an OS/2 dynamic link library (.DLL) that defines a set of functions callable from REXX programs. To use these functions, they must first be registered with the REXX interpreter using the RXFUNCADD function .

For example, when the OS/2 REXX interpreter is installed, an external function library called REXXUTIL is also installed. To ensure that the library is available to your program, add the following lines to the Init section of your project :

```
call RXFuncAdd 'SysLoadFuncs', 'REXXUTIL', 'SysLoadFuncs'
call SysLoadFuncs
```

Any of the REXXUTIL functions may then be used by your project.

Documentation for the REXXUTIL functions is available online as part of the **REXX Information** book installed in the VX-REXX folder. Help may also be obtained from the OS/2 command line by typing:

```
help rexx rexxutil
```

Other function libraries are also available from third party developers on a variety of electronic media.

# Using databases with VX-REXX

This chapter shows you how to use VX-REXX to create applications that use the REXX interface to IBM's Database 2 OS/2 (DB2/2) or Watcom SQL for OS/2. You should read this chapter if you have the Standard Edition of VX-REXX. If you have the Client/Server Edition, read the *Watcom VX-REXX for OS/2 Client/Server Edition* book instead. That manual describes the database objects, bound objects, and query editor tool that come with the Client/Server Edition. Contact Watcom for information about the Client/Server Edition.

Both DB2/2 and Watcom SQL have Application Programming Interfaces (APIs) for a number of languages including REXX. An application that has a REXX API provides routines that a REXX program can call to interact with that application. Any OS/2 software package that supports a REXX API can be used with VX-REXX.

Because VX-REXX projects are written in REXX, you can use VX-REXX to create database applications using the REXX API. The following sections show you how to build a sample application with which you can display and update information in an employee database. The complete program, **Employee database**, is included in the **Samples** folder contained in the VX-REXX folder.

This chapter assumes that you are familiar with database terminology and that you have DB2/2 or Watcom SQL installed. If you are using DB2/2, you need the sample database that comes with DB2/2. If you are using Watcom SQL, the application uses an equivalent database that comes with VX-REXX.

See IBM's *Database 2 OS/2 Guide* for information about installing the sample database for DB2/2. (To use the sample database you must first install it : enter **sqlsampl** at an OS/2 command prompt. When first installed, the sample database uses the userid **userid** and the password **password**).

## Overview

Using the REXX database API requires several steps, all of which are shown in this chapter:

oRegistering the API routines
oStarting the database manager (DB2/2 or Watcom SQL)
oOpening the database
oPreparing an SQL statement
oDeclaring a cursor
oRetrieving the data
oClosing the cursor
oClosing the database
oStopping the database manager

Some of these steps may be performed outside of your VX-REXX application if the application is just one of several that will use DB2/2 in an OS/2 session. These include registering the routines and starting and stopping DB2/2.

The program uses an employee database which has two tables: STAFF and ORG. The application retrieves several fields for all staff members and displays them in a window. Figure 85 shows the project's window.

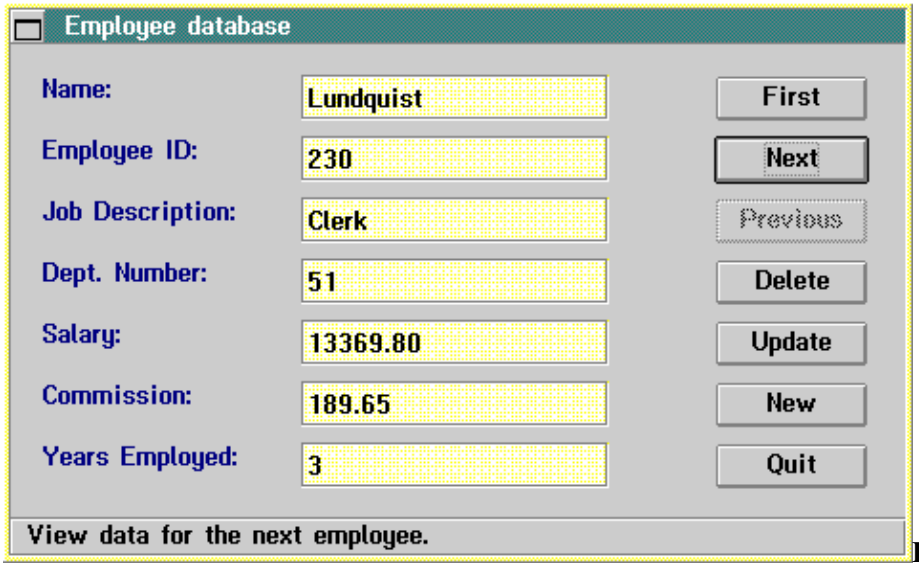


**Figure 85** Sample database application

The buttons on the window initiate some database action:

Button Description

First Display information for the first employee in the database.

Next Display the next employee.

Previous Display the previous employee. This button is enabled only if you are using Watcom SQL.

Delete Remove the current employee from the database.

Update Replace the information for the current employee with the values in the entry fields.

New Add a new employee to the database using values shown in the window.

Quit Close the database and terminate the application. Pressing this button is equivalent to double-clicking on the window's system menu.

## Registering routines

There are two REXX API routines that are used in this sample application :

SQLEXEC Used to process all SQL requests.

SQLDBS Used to call DB2/2 environment routines and utility routines.

There are other routines which provide access to the Query Manager but they are not used in this program. For a complete description of the REXX API refer to the IBM manuals *Database 2 OS/2 Programming Guide* and

*Database 2 OS/2 Programming Reference*, or the *Watcom SQL User's Guide*.

Before using these routines, you must register them using the RxFuncAdd routine. Once registered they remain in effect until they are dropped (with RxFuncDrop) or you shutdown OS/2.

The following code registers the functions. Note that the REXX code to register these routines is different for DB2/2 and Watcom SQL. The code assumes the variable **Globals.!DBtype** contains **WSQL** if you are using Watcom SQL.

```
call RxFuncDrop 'SQLDBS'
call RxFuncDrop 'SQLEXEC'

if( Globals.!DBtype = 'WSQL' ) then do
call RxFuncAdd 'SQLDBS', 'WSQLCAL2', 'WSQLDBSREXX'
call RxFuncAdd 'SQLEXEC', 'WSQLCAL2', 'WSQLEXECREXX'
end
else do
call RxFuncAdd 'SQLDBS', 'SQLAR', 'SQLDBS'
call RxFuncAdd 'SQLEXEC', 'SQLAR', 'SQLEXEC'
end
```

These statements are in the Init section of the **Employee database** sample program.

## Starting the database manager

You must start the database manager (DB2/2 or Watcom SQL) before you can access the database. The database manager processes the SQL statements that you issue in your program. In this sample program, the database is assumed to be installed locally and for a single user.

For both DB2/2 and Watcom SQL, the command to start the database manager is :

```
call SQLDBS 'START DATABASE MANAGER'
```

## Connecting to the database

Once the database manager is started you can connect to the database and access its tables. The REXX command to connect to the DB2/2 **sample** database is :

```
call SQLDBS 'START USING DATABASE sample'
```

while the command to connect to the equivalent Watcom SQL database is:

```
call SQLDBS 'START USING DATABASE db22samp USER DBA IDENTIFIED BY SQL'
```

The parameter **sample** (or **db22samp**) is the name of the database that you are using.

These statements appear in the Init section, after the REXX API routines are registered.

## Database API return codes

SQLEXEC and SQLDBS set predefined REXX variables each time they are executed . One of these is a compound variable SQLCA (SQL Communication Area). The database returns values to an application program through the set of variables defined in the SQLCA variable structure. The variable SQLCA.SQLCODE contains the primary error code. If the command is successful, the return code is 0. A positive error code indicates a warning condition and a negative error code indicates an abnormal condition.

It is good programming practice to check the error code after each database call to ensure that the command completed successfully. For example, if the database manager cannot be started, your application should inform the user that something is wrong rather then blindly proceeding. The following statements make this check:

```
call SQLDBS 'START DATABASE MANAGER'
if( SQLCA.SQLCODE \= 0 & SQLCA.SQLCODE \= -1026 ) then do
call VRMessage VRWindow(), ,
'Could not start the database manager:', ,
'SQLCODE =' SQLCA.SQLCODE, 'Error'
return
end
```

If the database is started successfully the return code is 0. The code -1026 is returned if the database is already started. Note that a comma at the end of a line is the REXX line continuation character.

The VX-REXX function VRMessage displays the error code.

## Preparing the SELECT statement

An SQL query is readied with an SQL PREPARE statement. The sample program prepares and issues a SELECT statement to retrieve the name, id, job, department name, salary, commission, and length of employment for each staff member in the organization .

In the following code, the first line defines the SQL query and assigns it to **stmt**. The second line readies the SQL request.

```
stmt = 'SELECT name, id, job, dept, salary, comm, years',
'FROM staff',
'ORDER BY name'
call SQLEXEC 'PREPARE s1 FROM :stmt'
```

The colon before **stmt** tells the database that this is a REXX variable.

If there are errors in your SQL query, the PREPARE command will not complete successfully so you should check the SQLCA.SQLCODE. When you are creating a program with a number of SQL queries you may want to test them using DB2/2's Query Manager or Watcom SQL's ISQL before coding them in your program.

## Declaring a cursor

You must declare a cursor before you can access the database tables. Declaring a cursor involves reserving a predefined cursor-name statement-name pair for the SELECT statement. A cursor is used to retrieve rows one at a time from the results of a query. The cursor names and statement names are always the letters **C** and **S** followed by an integer from 1 to 100. The integer for the cursor and statement should correspond. The pair are reserved with the SQL DECLARE statement :

```
call SQLEXEC 'DECLARE c1 cursor for s1'
```

You can use up to 100 cursor-statement pairs so that you can have up to 100 SQL queries active at any one time.

In the sample program the cursor is declared in the StartQuery section.

## Retrieving the data

After the SELECT statement has been prepared and the cursor has been declared , you can retrieve the results of the query one row at a time. The SQL OPEN command positions the cursor at the first row of the result table. You can then retrieve the values and assign them to REXX variables using the SQL FETCH command.

The following code opens cursor **C1** and reads data for the first employee into a number of REXX variables.

```
call SQLEXEC 'OPEN c1'
call SQLEXEC 'FETCH c1 INTO',
':Globals.!Emp.!name :ind.1,',
':Globals.!Emp.!id :ind.2,',
':Globals.!Emp.!job :ind.3,',
':Globals.!Emp.!dept :ind.4,',
':Globals.!Emp.!salary :ind.5,',
':Globals.!Emp.!comm :ind.6,',
```

```
':Globals.!Emp.!years :ind.7'
```

This code appears in the Fetch routine in the sample program.

In the **Employee database**, the values assigned to the REXX variables by the FETCH command are then displayed in the fields on the window.

## Displaying results

In the sample program the **Next** click event displays the results of the FETCH instruction in the entry fields. The VRSet function assigns the values from the REXX variables to the entry field values.

```
call VRSet 'EF_Name', 'Value', Globals.!Emp.!name
call VRSet 'EF_Id', 'Value', Globals.!Emp.!id
call VRSet 'EF_Job', 'Value', Globals.!Emp.!job
call VRSet 'EF_Dept', 'Value', Globals.!Emp.!dept
call VRSet 'EF_Salary', 'Value', Globals.!Emp.!salary
call VRSet 'EF_Comm', 'Value', Globals.!Emp.!comm
call VRSet 'EF_Years', 'Value', Globals.!Emp.!years
```

## Closing the cursor

When you are done making changes to a cursor, you must close it. The following statement closes cursor **c1**:

```
call SQLEXEC 'CLOSE c1'
```

If you want to make the changes to the database permanent, you must commit them. Alternatively, you can throw them away. The following code asks the user whether the changes should be saved, then executes a COMMIT or ROLLBACK command , depending on the user's choice.

```
button.0 = 2
button.1 = '~Save'
button.2 = '~Discard'
rc = VRMessage( VRWindow(),,
'Changes have been made to the database.' ||,
'Do you want to save or discard them?',,
'Employee database',,
'Q', 'button.', 1, 2 )
if( rc = 1 ) then do
call SQLEXEC 'COMMIT'
end
```

```
else do
call SQLEXEC 'ROLLBACK'
end
```

Code similar to that shown here appears in the Quit routine of the **Employee database** sample.

## Stopping the database manager

Before you close the application you should disconnect from the database and then stop the database manager. The following code does this.

```
call SQLDBS 'STOP USING DATABASE'
call SQLDBS 'STOP DATABASE MANAGER'
```

Code similar to that shown here appears in the Quit routine of the **Employee database** sample.

## Where to go from here

Combining the power and versatility of VX-REXX and SQL, you can create complex database programs using the basic steps outlined in this chapter.

You can also create database applications using the database objects in the Client/Server Edition of VX-REXX. With the Client/Server Edition, you can bind VX- REXX objects directly to database queries so that their values are set automatically from the database. The Client/Server Edition also comes with a design tool for constructing your SELECT statements using a visual point-and-click interface. Contact Watcom for more information about the Client/Server Edition of VX-REXX.

# Writing multithreaded applications

REXX programs are made up of one or more files. In the programs presented earlier in this book, only one file ran at a time. While one file may call another, the first file waits while the second one runs, so that only one routine is being run at a time. These types of programs are called *single threaded*.

Your REXX program can take advantage of VX-REXX multithreading capabilities by executing a number of files at once on different threads. These programs are said to be *multithreaded*.

All programs have at least one thread, called the *main thread*. The program can create more threads if required, although it is not a requirement to have more than one. Some guidelines on using threads are given later in this chapter .

Threads are not processes. A process is a programming unit that has its own memory and resources. A process may use threads to do its work in the same way that a company uses employees.

The rest of this chapter deals with specific topics related to programming with threads.

## Planning your application

If you want your application to perform some task in the background, you should use threads. If you want to make more than one window active at a time, you can also use threads, but it may be more convenient to use modeless windows instead .

Note that using threads does not necessarily make your program faster. Even though two REXX files may be executing at once, they still have to share the computer's CPU and hard drives.

When planning your program, you should design it with threads in mind, but you should not use threads right away. Debugging a single threaded program is much easier than debugging a multithreaded one because only one thing is happening at a time. Once you have debugged a single threaded version of your application, then make use of threads.

In most programs, a master-worker model for threads works well. In it, a master thread starts a number of worker threads which do some processing, then post a result back to the master thread. While it is possible to start threads from other threads, sticking to the master-worker model usually leads to code that is easier to develop and debug.

## Starting a code file thread

To start a code file as a thread, use the StartThread method on the Application object. Starting a thread is similar to calling a file: you give the name of the file and a list of parameters. The following code starts executing the file **File1** on a separate thread.

```
tid = VRMethod( 'Application', 'StartThread', 'File1' )
if tid = -1 then do
say 'Could not start thread.'
```

```
end
```

The method returns a *thread identifier*. The main thread of the application always has the identifier 0.

The maximum number of threads that can be active at any one time is 200. In addition, there are system limits defined in your OS/2 CONFIG.SYS file that provide an upper bound on the total number of threads that can be running across all processes.

## Starting a window file as a thread

To start a window file as a thread, you also use the StartThread method, but prefix the word **subroutine** to name of the window file and pass the value of VRWindow or a null string as the first parameter:

```
tid = VRMethod( 'Application', 'StartThread', ,
'subroutine Window2', VRWindow() )
```

The thread is started as a REXX *subroutine*, as opposed to a REXX *command*. The Main section of the window file uses this information and the value of the first parameter to assign the window's parent. If a null string is passed, the window's parent is the desktop. Otherwise the window is similar to a modeless secondary window, except that it is running on its on thread and as a separate file.

## Communicating with a thread

You can pass data to a thread in the following ways:

oParameters to the StartThread method

The optional parameters to the StartThread method are passed to the thread in the same way that they would if the file was simply called. For example,

```
call Worker 'Hello'
```

and

```
call VRMethod 'Application', 'StartThread', 'Worker', 'Hello'
```

and

```
call VRMethod 'Application', 'StartThread', 'subroutine Worker', ,
VRWindow(), 'Hello'
```

pass the string 'Hello' to Worker. In the first case, Worker is called synchronously. In the second, Worker is run as a code file on a new thread . In the third case, Worker is run as a window file on a new thread.

oPosting events to the master thread

A worker thread can post an event back to its master thread to report when it is finished. The thread can use the PostQueue method to do this.

For example, suppose Worker is a code-only file that counts the words in a text file whose name is passed as a parameter. The code for Worker is:

```
main:
parse arg filename

/* Count the words in a text file
*/
count = 0
do forever
text = Linein( filename )
if text == '' then leave
count = count + Words( text )
end
call Stream filename, 'C', 'Close'

/* Post the result back to the main thread
*/
call VRMethod 'Application', 'PostQueue', 0, 1,,
'call WorkerDone', 'WordCount', count

return
```

When Worker has counted all the words in the file, it posts the event ' call WorkerDone' to the main queue of the main thread. This event is executed in the main event loop, thus calling the routine named WorkerDone.

The WorkerDone routine uses VRInfo to retrieve the word count that was posted by the Worker thread:

```
WorkerDone:
count = VRInfo( 'WordCount' )
say 'There are' count 'words in the file.'
return
```

oGlobal data

Threads can use the PutVar and GetVar methods of the Application object to share information between threads. Since there is only one application object, access to global data is serialized.

## Getting information about running threads

Your program can get a list of running threads using the ListThreads application method. A thread can get its own identifier by invoking the GetThreadID method.

## Halting a thread

Use the HaltThread method to halt a thread. For example, the following instruction halts the thread whose thread identifier is in the variable **tid**:

```
call VRMethod 'Application', 'HaltThread', tid
```

Halting does not destroy the thread, but instead causes a REXX HALT condition to occur. Halt conditions are automatically trapped by the Halt section of a file. If the thread is blocked doing a lengthy operation (such as a database access), the halt condition is only raised when the threads resumes execution.

You cannot halt the main thread of your application.

# Controlling other programs

VX-REXX includes a number of facilities to enable your projects to control other programs. The following topics are covered in this chapter:

oRunning programs and executing OS/2 commands

oManipulating other programs' windows

oSending key strokes to programs

oCommunicating with programs via DDE

oWriting VX-REXX macros for other programs

## Running programs and executing OS/2 commands

Your project can run other OS/2 programs and commands using several methods .

### The address instruction

OS/2 programs and commands are executed from within a project using the **address** instruction:

```
address cmd 'dir'
```

Your project waits for the program or command to finish executing before continuing. The special variable **RC** holds the return value of the program or command .

To start a program or command without waiting for it to finish, use the OS/2 **start** command, as in:

```
address cmd 'start epm readme.txt'
```

The **start** command can also be used to start DOS and Microsoft Windows programs. For help on the OS/2 **start** command, type **help start** from an OS/2 command window.

### SysCreateObject and SysSetObjectData

Another way to start programs is to use the SysCreateObject and SysSetObjectData functions from the REXXUTIL external function library. To use these functions you must first load the library as described in the

'Extending VX¨REXX' chapter.

The following code opens a Workplace Shell object with id **<SOME_PROGRAM>** :

```
object = '<SOME_PROGRAM>'
setup = 'OPEN=DEFAULT'
call SysSetObjectData object, setup
```

See the online **REXX Information** for details on using the SysCreateObject and SysSetObjectData functions.

## Running commands that require OS/2 session VIO support

Since VX-REXX projects are Presentation Manager (PM) programs, you cannot directly execute OS/2 commands or programs which require VIO support. These are programs that expect to be running in an OS/2 windowed or full screen session. For example, the OS/2 PSTAT command uses VIO support. If you try to run it from REXX, you will get a 'SYS0436: An invalid VIO handle was found' error.

Use the OS/2 **start** command to run these programs. The following instructions will run PSTAT in its own OS/2 window:

```
address cmd 'start /win pstat <con >con'
```

The use of the redirection symbols **<** and **>** is explained below.

## Redirecting standard input, output and error

By default, VX-REXX redirects the standard input, output, and error streams to its console window. If you launch a program or execute an OS/2 command, the output is sent to the console and input is taken from the console.

For example, the following instruction sends a directory listing to the console:

```
address cmd 'dir'
```

When you run non-PM programs, the output is sent to the console unless you explicitly redirect it. The following instruction will run PSTAT in its own OS/2 window, and the results will be displayed in the window.

```
address cmd 'start /win pstat <con >con'
```

If the program you are running writes to standard error, you should add **2>con** to redirect standard error, as follows:

```
address cmd 'start /win pstat <con >con 2>con'
```

For more information on redirection, see the **Master Help Index** which is originally installed in the **Information** folder on your desktop. Read the ' Redirecting' and 'Input and output' sections.

# Manipulating windows

Presentation Manager (PM) windows (not to be confused with VX-REXX Window objects) can be used as arguments to VRGet, VRSet and VRMethod.

To identify a window, you must know its window handle. The window handle is a 4-byte hexadecimal value assigned to a window by PM. The window handle must be combined with the prefix **?HWND** before passing it to VX-REXX functions: for example, **?HWND08ef34**.

The console window is one exception to this rule: the name **Console** may be used to identify it.

## Finding a window

Window handles are obtained using several methods:

oUse the FindWindow method to find a window whose caption matches certain criteria.

oUse the ListWindows method to list all the top-level windows.

oUse the GetActiveWindow method to obtain the current active window.

oUse the GetFocusWindow method to obtain the current focus window.

oUse the HWnd property to obtain an object's window.

oUse the ServerHWnd property to obtain the handle to a DDE server.

oUse **?HWND1** for the root desktop window and **?HWND2** for the root object window .

Once a window handle has been obtained, the Parent, Sibling and FirstChild properties can be used to find the parent, sibling or child of a window. (If a null string is returned, no such window exists.)

All window handles are returned in the correct format for use with VRGet, VRSet and VRMethod.

## Setting window properties

Once you have a window's handle, you can use VRSet to set any of the following properties of that window:

Left The position of the window's left edge.

Top The position of the window's top edge.

Width The width of the window.

Height The height of the window.

Caption The contents of the window's title bar.

Visible Whether or not the window is visible.

Enabled Whether or not the window is enabled.

Parent The window's parent.

SiblingOrder (frame windows only) The next window in the z-order, if any.

WindowState (frame windows only) Whether the window is minimized or maximized .

WindowListTitle (frame windows only) The text that appears in the OS/2 Window List for this window.

Set these properties as you would for VX-REXX objects. For example, the following code looks for a window with the caption **OS/2 Window** and makes it visible :

```
hwnd = VRMethod( 'Screen', 'FindWindow', 'OS/2 Window', 'Desktop' )
if( hwnd \= '' )then do
call VRSet hwnd, 'Visible', 1
end
```

## Getting window properties

You can also use VRGet with a window handle to obtain information about a window. In addition to those properties listed above, you can also get the following properties VRGet:

ClassName The name of the window's PM window class.

ProcessID The ID of the process that created the window.

ThreadID The ID of the thread that created the window.

ID The window ID (a number from 0 to 65535).

Parent The parent window, if any.

Sibling The next window in the z-order, if any.

FirstChild The first child window, if any.

Owner The owner window, if any.

Self, HWnd The window handle.

Object The internal name of the VX-REXX object corresponding to the window handle, if any.

The following code uses GetFocusWindow to determine which object (if any) in your project has the input focus:

```
focus = VRMethod( 'Screen', 'GetFocusWindow' )
if( focus \= '' )then do
focus = VRGet( focus, 'Object' )
end
```

### Window methods

VRMethod may be used to invoke certain methods on windows:

oSetFocus

oMaximize (frame windows only)

oMinimize (frame windows only)

oRestore (frame windows only)

oClear (VX-REXX Console window only)

To set the focus to an arbitrary window, for example:

```
call VRMethod hwnd, 'SetFocus'
```

### Example programs

The **Window Controller** and **Hocus Focus** example programs in the **Samples** folder demonstrate the use of window handles.

## Sending keystrokes

Keystrokes may be sent to other Presentation Manager applications, and in some cases to windowed DOS and OS/2 applications, using the SendKeyString and PostKeyString methods of the Application object.

The syntax for SendKeyString is as follows:

```
call VRMethod 'Application', 'SendKeyString', object, keystring
```

The syntax for PostKeyString is similar:

```
call VRMethod 'Application', 'PostKeyString', object, keystring
```

For both methods *object* identifies the target object or window to which the keys are to be sent and *keystring* is the sequences of keystrokes to send. If a null string is specified as the target object for SendKeyString, the keystrokes will be sent to the window with the input focus.

Two things differentiate the SendKeyString and PostKeyString methods:

1.PostKeyString sends keystrokes to an object without waiting for them to be processed. These *posted* (queued) keystrokes are not translated into accelerator sequences. (Accelerator sequences are sequences of keystrokes that an application intercepts and handles before they can be passed to a window. Accelerators are usually used as shortcuts to activate menu items.)

2.SendKeyString sends keystrokes to an object one at a time, waiting until each is processed before sending the next. Each keystroke will also be checked for accelerator sequences.

If you *send* a keystring, a special sequence such as **{Alt}{F4}** may be handled as an accelerator, but not if it is *posted*.

## Keystroke syntax

Keystrokes are described using the same syntax that the KeyString property uses. To send the characters **A**, **B** and **C** in sequence, use the string **ABC**. To send a control character, prefix the character with the string **{Ctrl}**, as in **{Ctrl}Z** . The strings **{Alt}** and **{Shift}** are used to specify the **Alt** and **Shift** keys, as in **{Alt}H** or **{Shift}{F3}**. Function keys are specified as **{F1}**, **{F2}**, and so on .

You can combine modifiers:

```
{Ctrl}{Shift}J
{Alt}{Ctrl}{F10}
```

Modifiers only apply to a single keystroke. The following string is used to send the two keystrokes **Alt+F** and **O**:

```
{Alt}FO
```

A similar example sends **Alt+Z** followed by **Ctrl+X:**

```
{Alt}Z{Ctrl}X
```

There is no fixed limit on the number of keystrokes allowed in a single string, but you should keep the sequences fairly short.

## Controlling other applications

Keystrokes can be sent to objects within your own project or to windows in other applications. You need to know two things:

1.Decide which keystrokes are needed and whether they involve accelerators. If accelerators are needed, you must use SendKeyString.

2.Decide which object or window is to be sent the keystrokes. If you want to send keys to whichever window has the focus, use SendKeyString.

The first step requires that you run the application you wish to control and write down the series of keystrokes used to accomplish your task.

The second step requires some programming if you cannot simply assume that the destination window will be the window with the input focus. Use the methods outlined in the section above on finding window handles.

## Controlling a MultiLineEntryField

The MultiLineEntryField object can be controlled quite extensively by sending it keystrokes. The MLE understands the following keystrokes:

Key string Effect

{Del} Deletes the selected region or the character to the right of the cursor .

{Shift}{Del} Cuts the contents of the selected region to the clipboard.

{Ins} Toggles between insert and overstrike mode.

{Shift}{Ins} Paste the clipboard contents into the currently selected region .

{Backspace} Deletes the selected region or the character to the left of the cursor.

{Down}, {Up}, {Left}, {Right} These all move the cursor. Add a **{Shift}** character to select text as well.

{Ctrl}{Left}, {Ctrl}{Right} Move to the previous or next word.

{Ctrl}{Shift}{Left}, {Ctrl}{Shift}{Right} Select until the previous or next word.

{PageUp}, {PageDown} Move up or down a page.

{Ctrl}{PageUp}, {Ctrl}{PageDown} Scroll to the left or right.

{Home} Move to the beginning of the line.

{End} Move to the end of the line.

{Shift}{Home} Select from the current point to the beginning of the line.

{Shift}{End} Select from the current point to the end of the line.

{Ctrl}{Home} Move to the beginning of the data.

{Ctrl}{End} Move to the end of the data.

{Ctrl}{Shift}{Home} Select from the current point to the beginning of the data.

{Ctrl}{Shift}{End} Select from the current point to the end of the data.

A simple way to control an MLE in your own application is to define a routine such as this:

```
SendToMLE: procedure
call VRMethod 'Application', 'PostKeyString', arg(1), arg(2)
return
```

```
Call it as follows:

call SendToMLE 'MLE_1', '{Ctrl}{Home}'
```

```
Note that PostKeyString is used in case the control sequences the MLE
understands conflict with accelerators in your project.
```

## DOS and OS/2 Windows

Because of a limitation in OS/2 itself, only other Presentation Manager applications can be controlled by sending them keystrokes directly. Windowed DOS and OS/ 2 sessions can be sent some keystrokes, however, using the following method :

1.Copy the string of keys you wish to send into the clipboard using the PutClipboard method. Note that the special sequences **{Alt}**, **{Ctrl}**, and so on will not be translated. If you wish to send a special character, you must do the translation yourself, by finding the appropriate hexadecimal value for the character. For example, to add a **Ctrl+M** (carriage return) to the end of text (which is what pressing **Enter** typically does) you would do the following:

```
str = str || '0d'x
```

because **Ctrl+M** maps to **0D** hexadecimal.

2.Find the frame window of the DOS or OS/2 session. Use any of the methods previously discussed.

3.Send a message to the frame to update its menu in case the **Paste** option was previously disabled.

4.Send the string **p** to the frame window's system menu. The characters from the clipboard are now pasted into the session.

Here is a simple example of how to send the **dir** command to a DOS or OS/2 window, using the call:

```
call TypeInWindow win, 'dir' || '0d'x
```

Where TypeInWindow is defined as:

```
/* This function sends keystrokes to a DOS or OS/2 window
*/
TypeInWindow: procedure
parse arg frame, keys
if( VRGet( frame, 'ClassName' ) = 'WC_FRAME' )then do
call VRMethod 'Application', 'PutClipboard', keys
call PasteToWindow frame
end
return


/* Find the system menu and send it a 'p' to invoke the
paste menu item
*/
PasteToWindow: procedure
parse arg frame

child = VRGet( frame, 'FirstChild' )
do while( child \= '' )
if( VRGet( child, 'ClassName' ) = 'WC_MENU' & ,
VRGet( child, 'Id' ) = 32770 )then do
call VRMethod 'Screen', 'WinSendMsg', frame, 0x0033, ,
32770, child
call VRMethod 'Application', 'SendKeyString', ,
child, 'p'
leave
end
child = VRGet( child, sibling )
end
return
```

# Dynamic Data Exchange (DDE) client

DDE (Dynamic Data Exchange) is an OS/2 facility that allows applications to exchange data. Two applications engage in a DDE *conversation*, with one application being the DDE *client* (the one requesting the data), and the other a DDE *server* (the one responding to requests).

VX-REXX provides a DDE client, which allows your programs to talk to other applications, such as spreadsheets and word processors. It is possible, for example, to write a VX-REXX program that extracts data from a spreadsheet, then inserts that data into a word processing document.

DDE conversations always have a *topic*, which is defined when the client first connects to the server. The topic defines the context for the conversation. For example, when conversing with a word processor, the topic might be the file name of an open document. All DDE servers have a topic called **System** which can be used to obtain information about the server itself.

## Creating a DDE client

You create a DDE client the same way you create other VX-REXX objects: select its icon from the tool bar, then click on the window you are editing. The DDEClient looks similar to a DescriptiveText object. When your application runs, the DDE client updates its caption to describe its current state, a feature which is useful in debugging. When you are satisfied that your application is running correctly, you can make the DDE client invisible by setting its Visible property to 0 .

## Initiating a conversation

A DDE client can participate in only one conversation at a time. You establish a conversation to a DDE server using the Initiate and Accept methods. Initiate fills a stem variable with the list of servers and topics. Each compound variable contains a string of the form '*server*,*topic*'.

```
call VRMethod 'DDEC_1', 'Initiate', 'conv.'
do i = 1 to conv.0
parse var conv.i server ',' topic
say 'Conversation with' server 'on topic' topic
end
```

You can also initiate a conversation with a particular server and topic. The following code connects to server **Lotus**, topic **System.**

```
call VRMethod 'DDEC_1', 'Initiate', 'conv.', 'Lotus', 'System'
if conv.0 = 0 then do
call VRMessage VRWindow(), 'Could not connect'
end
```

At this point, no conversation has been established. The Accept method must be invoked to select a topic and server.

```
call VRMethod 'DDEC_1', 'Accept', 3
```

The optional parameter indicates the conversation to accept (as returned by _Initiate). If omitted, the first conversation is accepted.

## Executing server commands

To send strings to the DDE server for execution, use the Execute method. The syntax of the command string depends on the DDE server. Refer to the server application's documentation for a list of commands it will accept.

The Execute method returns **BUSY** if the server could not process the command . The following code fragment sends a command and resends it if the server is busy.

```
cmd = '[Type('Some text')]'
do forever
status = VRMethod( 'DDEC_1', 'Execute', cmd )
if( status \= 'BUSY' ) then leave
end
```

A **NoWait** option is available to send a command to the server without waiting for the reply. When the reply arrives, it will generate an Acknowledge event .

## Requesting data

Your application can request information from a DDE server using the Request and RequestList methods. Request sends a single request, while RequestList sends a number of requests using tab characters as delimiters.

All servers support the following requests:

Topics A list of topics for the server

SysItems A list of items available from the **Server** topic

Refer to the server documentation for details on valid item names.

The Request method is also used to create *warm links* and *hot links* between the client and the server. Hot and warm links notify the client using the _Notify event when a requested data item in the server has changed value. In a hot link (automatic notification), the new value is automatically sent to the client . In a warm link, the client fetches the new value from the server.

### Sending data

Data can be sent to a server using the Poke method. The server sets an item to a given value.

For example, the following instruction sets the value of cell B15 to 23.7 in a Lotus 1-2-3 spreadsheet:

```
call VRMethod client, 'Poke', 'B15', '23.7'
```

```
Refer to the server documentation for details on valid item names.
```

### Getting the status of a conversation

The DDE client stores information about the conversation in the following properties:

Status Indicates whether the conversation is still active

Server The name of the DDE server

Topic The conversation topic

Use VRGet to retrieve these properties at run time.

### Terminating a conversation

Use the Terminate method to end a conversation. A DDE client automatically terminates an existing conversation when it is destroyed or when a new conversation is initiated.

## Application macros

Some OS/2 applications such as the OS/2 Enhanced Editor use REXX as a macro language to extend the capability of the application.

The following example is a macro for the Enhanced Editor which displays the number of lines in a file.

```
/* A simple macro using the Enhanced Editor */
'extract /last'
lineCount = last.1
'sayerror File contains' lineCount 'lines'
```

```
The statements in quotation marks are commands which are sent to the
default command host -- in this case, the Enhanced Editor. The line
containing the word sayerror is an Enhanced Editor command which displays
```

the phrase 'File contains n lines' on the status line of the editor window.

Macros for the Enhanced Editor must be saved in a file with the extension ERX . For instance, the above example might be saved in the file named COUNT.ERX . To invoke the macro:

1.Start EPM.

2.Pick **Command dialog** under the (EPM) **Command** menu, or type **Ctrl**+**I** to open the command dialog.

3.Type **COUNT.ERX** to start the macro.

## Creating application macros with VX-REXX

You can use VX-REXX to create application macro files. These files are invoked by the target application. After the macro has been created within VX-REXX, you can make the application macro file which is invoked from within the host application.

You can create macro files from a VX-REXX project which contain only generic REXX instructions and commands. You can also create macro files which include the visual user interface capabilities of VX-REXX.

You cannot debug an application macro within VX-REXX since you must invoke the macro from within the host application. However, you can still use standard REXX debugging techniques which typically use the REXX **trace** statement and standard input and output. To simplify debugging a macro in a Presentation Manager session , include the VRRedirectStdIO function in your macro to redirect standard input and output to the VX-REXX console window.

To create a new application macro project:

1.Create a new project folder by dragging and dropping the VX-REXX Project template.

2.Open the project file within the new folder.

3.If no window is required, create a code-only project by selecting **New code file** from the **Project** menu.

4.Build the macro user interface if required, and enter the REXX instructions.

5.Create the application macro file by selecting **Make macro** from the **Project** menu.

You can now invoke the macro file from an application.

## Using application macro examples

The following examples are simple macros which you can create and run with the OS/2 Enhanced Editor. The procedures described here may differ for other application programs which use REXX as a macro language. For example, the Enhanced Editor requires that the REXX instructions be saved in a file with an extension of

ERX , such as TEST.ERX, instead of the standard CMD extension.

**Writing a macro with no window**

The following sample REXX instructions display the number of lines in a file using the Enhanced Editor sayerror command. To try this example, place these REXX instructions in the predefined routine Main of a code file only project.

```
/* */
'extract /last'
lineCount = last.1
'sayerror File contains' lineCount 'lines'
```

Assuming you make an application macro file named DISPLAY.ERX, you would type the following line in the Enhanced Editor's command dialog:

```
rx display
```

In this example, the macro would cause the phrase **'File contains n lines** ' to appear in the status area of the Enhanced Editor.

**Writing a macro with a window**

The following sample REXX instructions request the number of lines in a file and display this number in a VX-REXX window. To try this example, start a new project, then modify your Init routine to be the following:

```
Init:
window = VRWindow()

/* Set the window caption to the number of lines
*/
'extract /last'
information = 'file contains' last.1 'lines'
call VRSet window, 'Caption', information

/* Show the window
*/
call VRMethod window, 'CenterWindow'
call VRSet window, 'Visible', 1
call VRMethod window, 'Activate'
drop window
return
```

Assuming you make an application macro file named DISPLAY.ERX, the following line would be used to invoke the macro from the Enhanced Editor:

```
rx display
```

In this example, the macro would cause the phrase **'File contains n lines ** to appear as the title of the window.

When you try this macro, you may notice that clicking on the EPM window brings it to the front, covering the macro window. If you want to make the macro window modal to the application, you must disable the application window and set the owner of the macro window to be the application window. The following Init routine shows how to do this.

```
Init:
window = VRWindow()

/* Set the window owner to the EPM window
*/
epmWindow = VRMethod( 'Screen', 'GetActiveWindow' )
call VRSet window, 'FrameOwner', epmWindow

/* Disable the EPM window
*/
call VRSet epmWindow, 'Enabled', 0

/* Set the window caption to the number of lines
*/
'extract /last'
information = 'file contains' last.1 'lines'
call VRSet window, 'Caption', information

/* Show the window
*/
call VRMethod window, 'CenterWindow'
call VRSet window, 'Visible', 1
call VRMethod window, 'Activate'
drop window
return
```

When the user closes the macro window, the EPM window must be enabled. The following Fini routine shows how to do this.

```
Fini:
window = VRWindow()
call VRSet window, 'FrameOwner', ''
call VRSet epmWindow, 'Enabled', 1
drop window
return 0
```

# Creating objects at run time

This chapter explains how to create objects at run time and how to associate events to them.

## Using VRCreate and VRCreateStem

Any object may be created at run time by your project using the VRCreate and VRCreateStem functions. Objects created at run time are no different than objects created at design time.

The first argument to VRCreate and VRCreateStem identifies the parent object, either by its name (as returned by the Name property) or by its internal name (as returned by the Self property). For most objects this argument will be the name of the window or group box the object is to be a child of. A null string can only be used when creating primary windows.

The second argument to VRCreate and VRCreateStem identifies the type of object to create. For example, **PushButton** or **Window**.

The remaining arguments, if any, set initial property values. The following example uses VRCreate to create a ListBox:

```
listbox = VRCreate( VRWindow(), 'ListBox', ,
'Name', 'UserListbox', ,
'Visible', 1, ,
'Left', 200, ,
'Top', 200, ,
'Width', 600, ,
'Height', 1000 )
```

The same list box can be created using VRCreateStem as follows:

```
props.0 = 12
props.1 = 'Name'
props.2 = 'UserListbox'
props.3 = 'Visible'
props.4 = 1
props.5 = 'Left'
props.6 = 200
props.7 = 'Top'
props.8 = 200
props.9 = 'Width'
props.10 = 600
props.11 = 'Height'
props.12 = 1000

listbox = VRCreateStem( VRWindow(), 'ListBox', 'props.' )
```

Because external REXX functions can accept at most 20 arguments, VRCreateStem must be used in place of VRCreate when a large number of properties are to be set .

VRCreate and VRCreateStem both return the internal name of the new object or a null string if the object could not be created. The internal name can be used to refer to the object if the object's Name property was not set.

## Setting properties at creation time

All properties that can be set at run time using VRSet may be set when the object is being created. Certain properties can *only* be set when the object is being created and are 'read-only' after the call to VRCreate or VRCreateStem . The only way to change such a property after the object has been created is to destroy the old object and create a new one in its place.

The ListProperties method may be used to list which of an object's properties must be set at creation time only and which can be set at any time.

## Attaching events to an object

Event routines can be defined for an object created at run time by setting properties when the object is created or at any time after. For each event an object supports, a corresponding property exists and it is simply a matter of assigning the name of a REXX procedure to that property.

For example, to assign a Click event to a push button:

```
call VRSet pb, 'Click', 'call PushButton_Click'
```

The value of the property is a REXX **call** statement. When the push button is clicked, VX-REXX will call the PushButton_Click routine in the file. Note that REXX does not allow procedures to be defined at run time; you must define the procedures you wish to call in the design environment by creating new sections. Your procedures should use the VRInfo function to obtain the object name.

This technique may be used at run time to assign new event routines to any object.

## Events and event queues

Events are processed on a first-come, first-serve basis by calling the VREvent function and interpreting the string it returns. Events are kept in a queue , and VREvent removes the first event from the queue. The VRInit function creates a new event queue and makes it the current queue, while VRFini destroys the current event

queue and makes the previous queue the current one. VREvent always searches the current event queue. VRInit and VRFini are called automatically in the Main section of each window file.

Events are processed in the order in which they occur by repeated calls to VREvent in the Main section of the window file. An event that is being processed will not be interrupted by another event.

Each object is associated with an event queue. For Window objects, it is the event queue that was current when the object was created. For all other objects, it is the parent object's event queue. The objects associated with an event queue are destroyed when the queue is destroyed.

Events can be posted to a queue using the PostEvent and PostQueue methods .

## Destroying objects

Objects may be destroyed at any time using the VRDestroy function. Destroying an object destroys its child objects. After an object has been destroyed, it is no longer valid to refer to it.

# VX-REXX design time macros

You can extend the VX-REXX design environment by writing macros that are invoked from the object pop-up menu. You should familiarize yourself with the sections on creating application macros in 'Controlling other programs' before reading this section.

## The PROFILE.VRM file

The PROFILE.VRM file, located in the **Macros** folder, is executed as a macro by the VX-REXX design environment when it starts, before any project is loaded . You modify this file, using any text editor, to register the macros that are to be added to the object pop-up menu. VX-REXX includes a number of sample macros in the **Macros** folder -- simply uncomment the appropriate lines in the default PROFILE.VRM file and start VX-REXX to try them out.

## Search order

Except for the PROFILE.VRM file, which must be located in the **Macros** folder, the following search order is used to find macro files:

1.The directory where VX-REXX is installed, as returned by the VREPath function.

2.The SYSTEM subdirectories.

3.The MACROS subdirectory (which corresponds to the **Macros** folder).

4.The directories listed in the PATH environment variable.

Searching is only done for relative paths. If no extension is given, the extension .VRM is assumed.

When creating macros, it is recommended that you place them in the **Macros** folder.

## Installing macros

Macros are added to the object pop-up menu using the VREMacroAdd function :

```
call VREMacroAdd title, path, [list], [menu], [default]


The arguments are defined as follows:

title The title to use on the object pop-up menu.

path The name of the macro file to invoke. If a relative path is used,
the search order above is used to find the file.
```

list An optional list of object types, separated by semicolons. If specified, the macro will only appear on the pop-up menu if mouse button 2 is clicked on an object whose type matches one of the types in the list. If omitted, the macro always appears on the pop-up menu.

menu If **Open**, the macro appears in the **Open** conditional cascade menu on the pop-up menu. If omitted, the macro appears at the bottom of the pop-up menu .

default Set it to **Default** if the macro should be the default item in the conditional cascade menu, **NotDefault** if not. Only applies if *menu* is not null.

For example, the following example adds an item to the bottom of the pop-up menu:

```
call VREMacroAdd 'Match sizes', 'Resize'
```

The following example restricts the macro to certain object types:

```
types = ';PushButton;RadioButton;CheckBox'
call VREMacroAdd 'Button defaults', 'bdefault.vrm', types
```

The next example adds two items to the **Open**... conditional cascade menu:

```
call VREMacroAdd 'Tab editor...', 'settabs', 'Window',,
'Open', 'NotDefault'
call VREMacroAdd 'My settings...', 'myset', , 'Open', 'Default'
```

## Macro arguments

When a macro is invoked by VX-REXX, it is passed two internal object names as arguments. The first argument is the internal name of the object on which the pop-up menu was invoked (the object you clicked on with mouse button 2). The second argument is the internal name of the window on which the object resides. The VRWindow function works as usual within the macro. The VX-REXX design environment is disabled while the macro runs.

## Functions and methods

As a VX-REXX project, a macro has access to all the normal VX-REXX functions such as VRGet, VRSet, and so on. Several new methods and functions unique to the VX-REXX design environment are also available to the macro. See the MACROS .TXT file in the **Macros** folder for a list.

# Distributing your projects

## Run time library

The executable (EXE) and macro files built with VX-REXX require the run time library VROBJ.DLL to run. The OS/2 LIBPATH must include the directory that contains VROBJ.DLL.

Projects using one or more objects from an object library require the object library to run.

**Note:** The LIBPATH is defined in the OS/2 CONFIG.SYS file. You cannot change it from an OS/2 window. You must reboot OS/2 before any changes to the LIBPATH will take effect.

## Additional redistribution rights

Subject to the terms and conditions of the Watcom Software License Agreement, in addition to any Redistribution Rights granted therein, you are hereby granted a non-exclusive, royalty-free right to reproduce and distribute the VROBJ.DLL file located in the VX-REXX directory provided that (a) you not suppress, alter or remove proprietary rights notices contained therein; and (b) you indemnify, hold harmless and defend Watcom and its suppliers from and against any claims or lawsuits, including attorney's fees, that arise or result from the use or distribution of your software product.

Object libraries may or may not include additional redistribution rights -- consult their documentation for information.

# Samples

The following sections describe the sample programs included with VX-REXX . The files for these programs are located in subdirectories of the VXREXX\ SAMPLES directory. You can run any of these programs by opening the **Samples** folder and double-clicking on the sample's icon.

## Bounce

The **Bounce** program displays a picture of the Earth which bounces in a window . It illustrates a simple use of the Timer object. The position of the Earth is updated based on the timer's Trigger event.

To run the program, double click on the **Bounce** icon. To edit the project, double click on the **Bounce.VRP** icon in the **BOUNCE** folder.

## Button

The **Button** program displays four push buttons with the caption 'Push Me! '. When you click on any of the buttons, the button's color and caption changes to **Red**. When it is clicked again, the color and caption changes to **Yellow** . If you continue to click on a button, it will alternate between red and yellow .

The program demonstrates VRGet and VRSet, the two VX-REXX functions used to access object properties at run time.

To run the program, double click on the **Button** icon. To edit the project, double click on the **Button.VRP** icon in the **BUTTON** folder.

## Calculator

**Calculator** is a simple calculator program. It does addition, subtraction, multiplication, division, and percentages. The buttons CE and C clear the current entry and the current calculation, respectively. Memory buttons are provided to save (M+ and M-), recall (MR) and clear (MC) calculations.

The program illustrates using general routines for code that is used by many objects and shows how to pass arguments to a REXX routine.

The calculator sample also demonstrates the use of the KeyPress event and KeyString property. Instead of using the mouse to click on the calculator buttons you can just type the calculator operations on the keyboard. The C button is accessed using the letter C on the keyboard, the CE button through the letter E on the keyboard, and the memory functions by pressing **F5**-**F8** on the keyboard for M+, M-, MR, and MC respectively.

To run this program, double click on the **Calculator** icon. To edit the project, double click on the **CALC.VRP** icon in the **CALC** folder.

## DDE Explorer

The **DDE Explorer** program lets you explore the applications on your system that support the DDE interface. You can list the DDE servers and topics on the servers. After connecting to one of the servers, you can then request information, execute commands, and send data to the server.

To run this program, double click on the **DDE Explorer** icon. To edit the project, double click on the **DDE.VRP** icon in the **DDE** folder.

## DragDrop

The **DragDrop** demo illustrates the way you can add drag and drop to your programs. The program displays two containers, each with a number of records. You can drag records from one container to another. Information about the source and target records and containers is shown in the VX-REXX console window.

To run this program, double click on the **DragDrop** icon. To edit the project, double click on the **DRAGDROP.VRP** icon in the **DRAGDROP** folder.

## Employee database

The **Employee database** program uses the REXX API to DB2/2 and Watcom SQL to view and update an employee database. See the application was designed.

To run this program, double click on the **Employee database** icon. To edit the project, double click on the **Employee.VRP** icon in the **EMPLOYEE** folder.

## File browser

The **File browser** example is a simple file browser. It displays files and lets you navigate the directory tree much like the OS/2 Drive object does.

The program displays the contents of the current directory in a container . You can display the directory as icons or as text. You change directories by double clicking on a folder icon in the container.

To run this program, double click on the **File Browser** icon. To edit the project, double click on the **FILEBRO.VRP** icon in the **FILEBRO** folder.

## Hint and Help

The **Hint and Help** sample program demonstrates the use of hints and context sensitive help in VX-REXX projects. When the program is running, move the pointer over each of the objects in the window to view the hints for each. The hints appear in the status area in the bottom of the window. Press the **F1** key to open a help window.

To run this program, double click on the **Hint and Help** icon. To edit the project, double click on the **HINTHELP.VRP** icon in the **HINTHELP** folder.

## Mind Game

**Mind Game** is a game in which you must guess the colors of four buttons in ten guesses or less. When it is run, the buttons for the first guess are located above the horizontal bar at the bottom left hand corner of the window. To begin, choose a color from the color palette at the top right hand corner of the window. The currently selected color is shown in the large colored button to the right the color palette. When you have chosen a color, click on the button to which you want to assign the color. When you have chosen colors for all four buttons, click on **Guess**. An **X** is displayed for each button for which the color is correct . An **O** indicates that a correct color is chosen but it is not assigned to the correct button. You continue guessing until you guess correctly or complete ten tries. The **New** item in the **Game** menu starts a new game. The **Peek** item lets you see the correct solution. The **About** item in the **Help** menu shows copyright information.

This program shows how to add a dialog to a program (the About dialog) and how to use VX-REXX predefined functions (VRMessage). It also illustrates using general routines for code that is used by many objects and shows how to pass arguments to a REXX routine. It shows some of the features of the REXX language such as program control statements (**do** and **if**) and how to use REXX stem variables as arrays to simplify programming. **MindGame** also demonstrates the use of menus.

To run this program, double click on the **Mind Game** icon. To edit the project, double click on the **MINDGAME.VRP** icon in the **MINDGAME** folder.

## MMW

The **MMW** sample program demonstrates the difference between modal and modeless windows. It allows you to open both modeless and modal windows, and explains at each step how you should proceed.

To run this program, double click on the **MMW** icon. To edit the project, double click on the **MMW.VRP** icon in the **MMW** folder.

## Movies

The **Movies** sample program demonstrates the use of VX-REXX with the multimedia extensions to OS/2. You will require the IBM Multimedia Presentation Manager/2 software to run this sample. This software is included in OS/2 2.1 but is installed separately.

This sample allows you to play movies, either in the main thread, or as part of a separate thread, allowing you to continue working in the current thread.

To run this program, double click on the **Movies** icon. To edit the project , double click on the **Movies.VRP** icon in the **MOVIES** folder.

## Notebook

The **Notebook** program lets the user change the color and size of a push button at run time. It does this by displaying a notebook with the BackColor, ForeColor, Height, and Width properties. The user can alter the values and see the changes to the button.

This sample shows how to use the Notebook object in a simple application.

To run this program, double click on the **Notebook** icon. To edit the project, double click on the **Notebook.VRP** icon in the **NOTEBOOK** folder.

## Popup

The **Popup** program lets the user change the color and caption of a number of push buttons at run time. When the user clicks mouse button 2 on any of the buttons, a pop-up menu is displayed, listing the button properties that can be changed .

This sample shows how to use the Popup method to display a pop-up menu.

To run this program, double click on the **Popup** icon. To edit the project, double click on the **Popup.VRP** icon in the **POPUP** folder.

## Printing

The **Printing** sample program lets the user send a file to a given printer. It illustrates the use of the ListPrinters method and the VRPrintFile function .

To run this program, double click on the **Printing** icon. To edit the project, double click on the **Printing.VRP** icon in the **PRINTING** folder.

## RGB

The **RGB** project illustrates the use of the Slider and ValueSet set objects . The sliders control the color of each item in the value set.

To run this program, double click on the **RGB** icon. To edit the project, double click on the **RGB.VRP** icon in the **RGB** folder.

## Scan

In this sample, VX-REXX was used to create a REXX macro that can be run with the OS/2 Enhanced Editor (EPM). When run from EPM, the **Scan** macro searches the current file for all REXX labels (with format *labelname*) and displays the labels in a list box. If you double click on a label in the list box, that label becomes the current line in the editor.

The sample program was created with VX-REXX. The macro was created using **Make macro** from the VX-REXX **Project** menu.

To start the macro, open the **EPM with Scan** object in the **Scan** folder. Alternatively, you can run the macro from within EPM by doing the following:

1.Press **Ctrl+I** or choose **Command dialog** from the **Command** menu.

2.Type the following in the entry field:

**rx c:\vxrexx\samples\scan\scan**

The **Scan** folder also includes a **Profile.ERX** file that will add a menu to EPM which can be used to run the macro. To use the profile:

1.Open EPM.

2.Select **Command dialog** from the **Command** menu.

3.Enter **Profile on** and select **OK** in the **Command dialog**.

4.Select **Save options** from the **Options** menu.

5.Close EPM.

Now when you double click on the **EPM with Scan** icon and select a label, EPM should include a **Labels** menu. Selecting **Find** from this menu will run the sample macro.

To edit the project, double click on the **Scan.VRP** icon in the **SCAN** folder .

For more information on writing macros for other programs, see the ' Controlling other programs' chapter in this manual.

# Threads

The **Threads** sample program demonstrates VX-REXX's ability to create multithreaded applications. This feature allows you to write applications which perform lengthy operations in the background. For example, an application could allow the user to continue to work in the main window while saving files in the background . The main window would be one thread, and the part of the program saving the files would be another thread.

This sample allows you to create threads by clicking on the **Spawn New Thread** button. A new window is opened into which you can enter text. Clicking on the **Send to parent** button then sends the text to the main window, where it is displayed in a list box.

You can also stop the new threads that you create by selecting a thread number, then clicking on the **Halt Selected Thread** button.

To run this program, double click on the **Threads** icon. To edit the project, double click on the **Threads.VRP** icon in the **THREADS** folder.

# Window Controller

The **Window Controller** demo illustrates the use of the ListWindows method and PM window handles. The program lists all of the top level frame windows that are currently visible on the desktop.

Some of these windows may have a '+' next to their name. By clicking on a plus sign, you expand the display to show the child windows of the selected window. Some of these child windows may have child windows of their own, and you can expand the display to show them as well. When a window's children are displayed, the plus sign changes to a minus sign. Clicking on this minus sign hides the child windows of the selected window.

After selecting a window in the program's container, you can press buttons to minimize, maximize, restore, or shake the selected window. There is also an update button, which updates the list of windows displayed in the container .

To run this program, double click on the **Window Controller** icon. To edit the project, double click on the **WinCtrl.VRP** icon in the **WINCTRL** folder.

**Contents**

**Index**

**Index**

**Contents**

- Introduction
  - ♦ What you should know before starting
  - ♦ How to use this manual
    - ◊ Organization
    - ◊ Syntax conventions
  - ♦ Online information
    - ◊ Online books
    - ◊ REXX information
    - ◊ Context help at design time
  - ♦ Technical notes
  - ♦ Other sources of information
- Setting up
  - ♦ Before you start
    - ◊ Check your VX-REXX package
    - ◊ Check your machine
    - ◊ Installing OS/2 REXX, the Enhanced Editor and the resource compiler
  - ♦ Installing VX-REXX
  - ♦ Send in your registration card
  - ♦ Watcom VX-REXX folder contents (Standard Edition)
    - ◊ VX-REXX
    - ◊ Read Me First
    - ◊ VX-REXX Programmer's Guide
    - ◊ VX-REXX Reference